Semántica y toma de decisiones en Internet de las Cosas

Semantics and decision making on the Internet of Things

Castor Miguel Pérez Melián
Departamento de Ingeniería Informática
Escuela Técnica Superior de Ingeniería Informática

Trabajo de Fin de Grado

La Laguna, 09 de Julio de 2014

D. **José Ignacio Estévez Damas**, con N.I.F. 43.786.097-P profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática de la Universidad de La Laguna

C E R T I F I C A

Que la presente memoria titulada:

"Semántica y toma de decisiones en Internet de las Cosas"

ha sido realizada bajo su dirección por D. Castor Miguel Pérez Melián, con N.I.F. 54.060.271-J.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 9 de julio de 2014

Resumen

El objetivo de este proyecto es el desarrollo de un prototipo de aplicación que nos permita monitorizar y controlar de manera centralizada la toma de decisiones de una cantidad arbitraria de sistemas de sensado con componentes distribuidos (MDSS), mediante una especificación en una lógica de primer orden basada en la descripción de estados con flujos. Para ello se ha adaptado una disciplina utilizada para describir, monitorizar y, finalmente, jugar juegos por turnos llamada General Game Playing (GGP).

La idea del prototipo es la de un servidor central que aplique técnicas utilizadas en GGP, normalmente usadas en el control y monitorización de juegos estratégicos, a sistemas interconectados en Internet de las Cosas de modo que, dadas ciertas condiciones, el intercambio de información entre dispositivos se pueda realizar de manera controlada, reactiva y homogénea, y dando pie a la utilización de lenguajes basados en predicados lógicos que proporcionan representaciones compactas de las reglas que deben seguir los sistemas automáticos.

Se trata de un proyecto experimental con la intención de realizar una demostración práctica de las ventajas que ofrece el uso de la disciplina GGP, y por extensión, su lenguaje de representación de reglas, GDL, en este tipo de entornos en los que no son utilizados comúnmente. Además de esto, se estudiará una prueba de concepto sobre la utilización de las Redes Lógicas de Markov (MLN) para la detección de posibles fallos en sistemas automáticos, utilizando inferencia para intentar distinguir entre distintos tipos de errores o problemas que se pueden dar en los diversos componentes de un sistema distribuido.

Este documento pretende ofrecer al lector una visión completa de cómo se

han intentado adaptar este tipo de tecnologías y técnicas al prototipo que se desea implementar, e intentar llegar a algunas conclusiones sobre su posible uso más allá de los entornos experimentales y de laboratorio.

Palabras clave

Internet de las Cosas, Aplicaciones distribuidas, General Game Playing, Game Description Language, Ideas preliminares

Abstract

The main goal of this project is to develop an application prototype that allows us to monitor and control the decision making of an arbitrary number of Modifiable Distributed Sensor Systems (MDSS) in a centralized manner, through the specification of a first order logic based on the description of states with fluents. To that end we've adapted the usage of General Game Playing, a discipline used for defining, monitoring and, finally, playing turn-based games.

The main idea of this prototype is to use a central server that applies some techniques utilized by GGP, usually intended for controlling and monitoring strategic games, to interconnected systems in the Internet of Things so we can, given certain conditions, exchange information between devices and do it in a controlled, reactive and homogeneous way, and giving the opportunity to use logic predicate based languages that provide compact representations of the rules that automatic systems must follow.

It's an essentially experimental project with the goal of proving, in a practical manner, the advantages of the usage of GGP and its rules representation language, GDL, in this kind of environment where they're not usually utilized. We'll also study the idea of using Markov Logic Networks (MLN) to detect failures on automatic systems, applying inference in an attempt to distinguish between different kinds of errors that can appear on any given component of a distributed system.

This document is intended to give the reader a complete insight on how we've tried to adapt this kind of technologies and techniques to the desired prototype, and trying to shed some light on their usage outside an experimental, and thus, completely controlled, environment.

Keywords

Internet of Things, Distributed applications, General Game Playing, Game Description Language, Preliminary ideas

Índice General

Índice de figuras

Índice de tablas

Capítulo 1. Introducción

1.1 Contexto del trabajo

1.1.1 Internet de las Cosas

El término "Internet de las Cosas" (Internet of Things, IoT), atribuido a Kevin Ashton [1], describe un conjunto de objetos (elementos, cosas) reales unívocamente identificables y su representación virtual en una estructura organizada e interconectada de manera similar a Internet, permitiendo así la utilización, gestión y creación de servicios, la capacidad de compartir información y la interacción directa, mediante sensores y actuadores de distintos tipos, con el usuario y el ambiente que lo rodea.

Citando a Ashton, también en [1]:

"*Si tuviéramos ordenadores que supieran todo lo que tuvieran que saber sobre las "cosas", mediante el uso de datos que ellos mismos pudieran recoger sin nuestra ayuda, nosotros podríamos monitorizar, contar y localizar todo a nuestro alrededor, de esta manera se reducirían increíblemente gastos, pérdidas y costes. Sabríamos cuándo reemplazar, reparar o recuperar lo que fuera, así como conocer si su funcionamiento estuviera siendo correcto. El Internet de las Cosas tiene el potencial para cambiar el mundo tal y como hizo la revolución digital hace unas décadas.*"

Por lo tanto, el paradigma que rodea el proyecto, y que le da su razón de ser, es el auge de esta relativamente nueva manera de comprender las interacciones tecnológicas entre dispositivos de forma ubicua, automática e

instantánea, y que tiene la finalidad de hacer más fácil y cómodo el día a día del ser humano.

El trabajo que nos ocupa se centra sobre todo en la tecnología semántica aplicada a Internet de las Cosas, comprendiendo la gestión, identificación y representación de la gran cantidad de información que pone en movimiento este paradigma. Específicamente, se dará gran importancia a los procesos de monitorización y control de la toma de decisiones de los sistemas asociados a una red IoT, haciendo el enfoque del trabajo mayoritariamente orientado a la semántica y la toma de decisiones, como el propio título indica.

1.1.2 Monitorización y toma de decisiones

Como se ha mencionado previamente, la monitorización aplicada a sistemas interconectados en Internet de las Cosas es uno de los temas centrales a tratar en el trabajo. Por dar una definición, la monitorización, como término tecnológico, consiste en la vigilancia y recopilación de datos, en nuestro caso, de diversos sistemas automáticos, de manera automatizada para su uso en análisis y toma de decisiones.

Uno de los mayores desafíos a la hora de aplicar monitorización y control de procesos en el paradigma de Internet de las Cosas es la enorme cantidad de dispositivos interconectados entre sí, y, como se mencionó brevemente antes, la cantidad de información que se mueve entre los mismos. Por lo tanto, la comunicación entre procesos, sobre todo los que aporten información hacia y desde un dispositivo de monitorización, debe ser tratada con especial cuidado para que la información que se transmita sea concreta, breve y semánticamente correcta por todas las partes implicadas.

Así pues, la búsqueda de un lenguaje común para la representación de la información, de manera lo más compacta y concreta posible, nos lleva a la

utilización de un lenguaje basado en predicados lógicos, llamado GDL, del que hablaremos en secciones posteriores de esta memoria, para realizar la representación de las reglas y la información a intercambiar, y su posterior análisis con herramientas externas, acercándonos así a un punto en que la toma de decisiones y la monitorización se puede realizar de la manera más homogénea y semánticamente transparente que se pueda.

1.2 Objetivos y antecedentes del trabajo

1.2.1 Objetivos del trabajo

El objetivo de este proyecto es producir un sistema automático cuyos componentes y reglas se expresen en un lenguaje de alto nivel que dé lugar a representaciones compactas. Además de los beneficios derivados de esta clase de lenguajes en cuanto a facilidad de comprensión del sistema y por lo tanto de depuración del código, se pretende que la creación de subsistemas de monitorización para la detección de fallos sea lo más sencilla posible.

El ámbito de aplicación del proyecto son los sistemas de sensado con componentes distribuidos y modificables. Un sistema de sensado con componentes distribuidos y modificables (Modifiable Distributed Sensor System, MDSS) está formado por un conjunto de componentes reconfigurables de captación de información del entorno unívocamente identificables. Por tanto, algunas de las funciones típicamente realizadas por estos componentes son: detección de eventos, sensado de magnitudes físicas del entorno, identificación unívoca, reconfiguración y tareas de procesamiento y comunicación de la información. Desde este punto de vista los componentes de un MDSS pueden ser muy variados (por ejemplo un computador convencional, una etiqueta RFID, un código de barras, un nodo perteneciente a una red de sensores). Es también muy importante resaltar que el objetivo del MDSS es captar información útil para un objetivo determinado por lo que

debe entenderse como parte de un sistema integrado por elementos adicionales que permiten solucionar un problema de automatización.

En la búsqueda de un lenguaje suficientemente simple, pero con gran capacidad de representación nos hemos fijado en la disciplina dentro de la Inteligencia Artificial denominada "General Game Playing" (GGP). Esta disciplina, que pretende la creación de algoritmos capaces de enfrentarse al reto de aprender a jugar, no un único juego, sino cualquiera representable en un lenguaje base, nos interesa por la metodología empleada y precisamente por su lenguaje de representación de los juegos.

Se trata por tanto de un proyecto experimental, donde se pretende estudiar la aplicación de la metodología y representación de problemas empleada en la disciplina GGP en el campo de la automatización basada en sistemas de sensado con componentes distribuidos.

En resumen, este proyecto requerirá:

- El estudio de la metodología y lenguaje de representación en la disciplina GGP.
- La adaptación de la arquitectura y lenguaje de representación en GGP al problema de la automatización.
- La implementación de un prototipo que permita realizar las tareas básicas de automatización expresadas en el lenguaje de GGP y con su arquitectura de referencia.
- La discusión y puesta en práctica de tareas de monitorización, al menos como prueba de concepto, con el fin de analizar las ventajas e inconvenientes de la técnica propuesta.

1.2.2 Solución propuesta

Para intentar cumplir con todos los requisitos planteados en el proyecto del Trabajo de Fin de Grado, se propondrá una solución experimental de cara a realizar una demostración y un estudio prácticos sobre la viabilidad y adecuación de la aplicación de la disciplina General Game Playing al entorno tecnológico y conceptual de Internet de las Cosas.

Como se ha mencionado en los objetivos, la solución que se va a desarrollar consiste en el diseño e implementación de un prototipo que permita la realización de distintas tareas que nos permita la adaptación de la arquitectura de GGP a el uso de sistemas automáticos con componentes distribuidos, incluyendo el control sobre el estado de los sistemas, sobre la toma de decisiones y sobre las posibles incoherencias que pueden surgir culpa de errores físicos, ya sean en el funcionamiento de algún dispositivo o un fallo humano.

Sin entrar en detalles profundos sobre el diseño y la arquitectura interna de la solución a implementar durante la duración de este proyecto, ya que ese será el tema principal del capítulo 3, pasaremos a continuación a describir la solución planteada, resumida en el siguiente esquema:

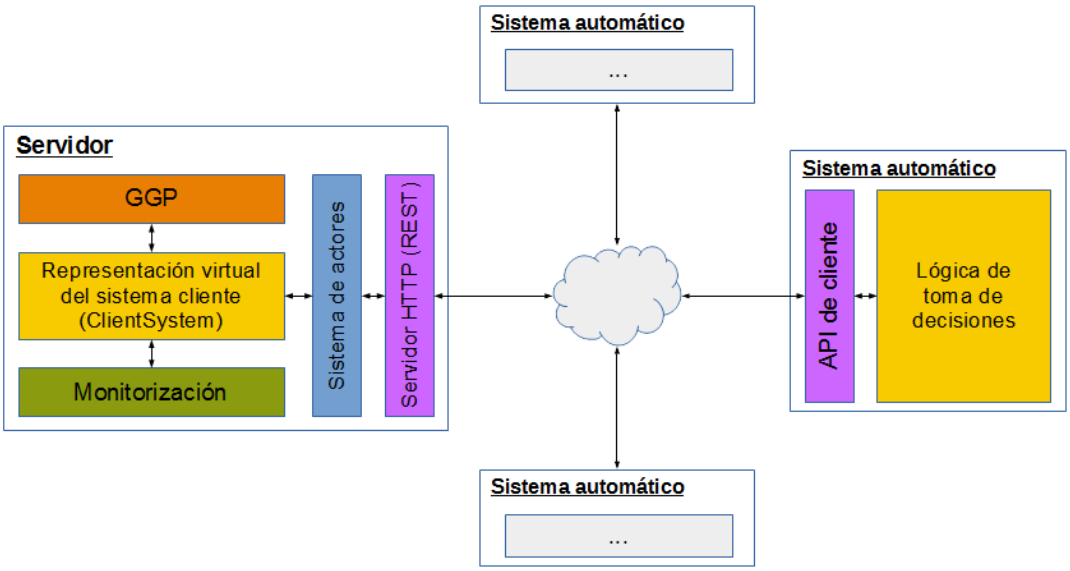

Figura 1.1. Estructura general del prototipo

En el diagrama podemos observar que el prototipo se basa principalmente en dos elementos que agrupan todos los conceptos que necesitamos desarrollar para llevar a cabo los objetivos del proyecto y que conforman, de manera general, la solución propuesta en su fase de prototipo. Se trata de una arquitectura cliente-servidor, muy habitual hoy en día, de tal forma que los sistemas automáticos, que son los que poseen el conocimiento para tomar decisiones, son los clientes, y se conectan mediante una API de peticiones HTTP a un servidor central.

Como vemos también en el esquema, pueden existir varios sistemas automáticos, no necesariamente iguales, pero con la misma estructura: deben contener la API de conexión al servidor y el código que les permita tomar decisiones con la información obtenida del servidor, más la suya propia.

Del lado del servidor, sin entrar demasiado en detalles, se pueden destacar tres partes fundamentales, de derecha a izquierda: la capa de conexión HTTP, para recibir las peticiones que los clientes realicen y devolver la información necesaria para garantizar el flujo de la información por ambas partes; el

sistema de actores, que describiremos con más profundidad más adelante, es el que, a grandes rasgos, se encarga de mantener unidas las tres capas principales de la aplicación, de manera interna; por último la capa del núcleo, los tres elementos representados a la izquierda: GGP, Monitorización y la representación virtual de los clientes (sistemas automáticos), *ClientSystem*. El núcleo es, como podemos inferir, el centro de operaciones de la lógica de negocio: es el que se encarga de almacenar la información relevante sobre los sistemas cliente, el que se encarga de procesar el estado del juego de GGP por cada cliente y el que almacena la información necesaria para que la monitorización e inferencia de errores del sistema pueda ser una realidad, aunque sea de manera experimental.

Este proyecto centrará casi todo su peso en la parte del servidor, que es la que nos interesa para realizar el estudio práctico del uso de este tipo de tecnologías aplicadas a la toma de decisiones en sistemas automáticos, ya que la lógica propia de la toma de decisiones tiene una casuística muy variada y no es la finalidad de este trabajo. De este modo podemos centrar nuestros esfuerzos en la implementación de un servidor general, que pueda ser utilizado para monitorizar cualquier tipo de sistema automático que establezca unas reglas basadas en el lenguaje de GGP y disponga de una API para realizar la conexión.

1.2.3 Antecedentes

El uso de lenguajes lógicos en entornos de automatización no es un concepto nuevo. Ya en el artículo [2], por aportar algo de visión teórica sobre el tema, se define GDL, el lenguaje utilizado por GGP, como "un lenguaje de especificación para una gran variedad de entornos multiagente", una definición que va bastante más allá de su aplicación original para la descripción de juegos y abre las puertas a su posible uso para la especificación

de las reglas de funcionamiento de dichos entornos. Un entorno multiagente discreto, síncrono y determinista constaría de los siguientes elementos:

Un conjunto finito de agentes, que sería un subconjunto del conjunto de expresiones simbólicas: $R \subseteq \Sigma$
Un estado inicial, que sería un subconjunto finito del conjunto de expresiones simbólicas: $s_l \subseteq \Sigma$
Un conjunto de estados terminales que sería un subconjunto de los posibles subconjuntos del conjunto de expresiones simbólicas: $t \subseteq 2^{\Sigma}$
Un subconjunto formado por tripletas de la forma (r, exp, s), donde r sería un agente, exp sería una expresión simbólica (que representaría una acción) y s un estado. Este subconjunto representa la legalidad de la acción exp para el agente r en el estado s: $l \subseteq R \times \Sigma \times 2^{\Sigma}$
Una función de transición entre estados que permite obtener el nuevo estado tras aplicar los agentes cada uno una acción: $u : (R \to \Sigma) \times 2^{\Sigma} \to 2^{\Sigma}$
Una función de utilidad formada por tripletas de la forma (r, exp, s) donde r representa un agente, n es un número natural (utilidad) y s es el estado: $g \subseteq R \times \aleph \times 2^{\Sigma}$

Esto significa, sintetizando, que un entorno multiagente será aquel que contenga un conjunto finito de agentes, un estado inicial y un conjunto de estados terminales, y que implemente una función de transición que permita, mediante la aplicación síncrona de acciones legales asignadas a cada uno de los agentes del entorno, pasar de un estado a otro.

El lector observará, a lo largo de este documento, que estamos tratando, efectivamente, con entornos multiagente, ya que utilizaremos el lenguaje GDL para definir las reglas por las que se rigen los MDSS y modelarlos como un entorno multiagente general.

Además de estos antecedentes teóricos, hay que añadir que el lenguaje de representación elegido para trabajar en este proyecto ha sido utilizado para algunos estudios y demostraciones en el ámbito de la robótica, como [3] y [4]. Este último trabajo dio lugar a un sistema de razonamiento capaz de usar GDL para razonar sobre un juego [5]. Esto demuestra la viabilidad y la bondad de esta representación para ser utilizado por razonadores automáticos, que es uno de los objetivos de la investigación que pretende desarrollarse a partir de este proyecto.

1.2.4 Conceptos

A modo de prefacio para los capítulos más específicos de este documento, consideramos adecuado aclarar y definir una serie de conceptos a los que esta memoria se referirá en diversas ocasiones, a fin de evitar confusiones y dejar claros los términos que vamos a utilizar.

- **Juego** (Game): un juego es un elemento, almacenado del lado del servidor, que encapsula la información sobre las reglas de actuación de un sistema automático o cliente en general. Los juegos son la base de todo el modelo de GGP, y utilizaremos el término juego indistintamente para referirnos a juegos convencionales (a la hora de poner ejemplos), tanto como a la relación interactiva entre los sistemas automáticos cliente y el servidor central.
- **Partida** (Match): Una partida consiste en la ejecución de una instancia particular de un juego desde el estado inicial hasta un estado terminal. Este término no será utilizado a lo largo de esta memoria, ya que la adaptación de la arquitectura de GGP a los sistemas automáticos no contempla el concepto de partida. El razonamiento detrás de esto es que una partida es un concepto eminentemente finito y acotado, con una meta

y reglas de finalización en las propias hojas de reglas, mientras que las hojas de reglas de los sistemas automáticos con los que se ha trabajado como muestra no especifican ninguna meta ni condición de finalización, ya que se trata de sistemas que están continuamente funcionando, sin la necesidad de interacción con un agente humano. Sin embargo, merece la pena mencionar este concepto para posibles futuras referencias que puedan hacer uso de esta memoria.

- **Jugador** (Player): denominaremos jugador a aquella entidad de cualquier tipo capaz de, dado un estado del juego y un conjunto de movimientos legales, realizar una toma de decisiones y seleccionar los movimientos que se estimen oportunos mediante cualquier tipo de algoritmo de decisión y trasladar dicha decisión al juego mediante su rol en el mismo. El tipo de algoritmo de decisión es irrelevante para el desarrollo del trabajo, y se puede utilizar cualquier tipo, ya sea selección aleatoria, arbitraria, algún tipo de inteligencia artificial, etc. Es particularmente importante no confundir el concepto de jugador con el de rol.
- **Rol** (Role): un rol es el papel que cumple un jugador de cara a un juego. El rol de un jugador puede ser fijo o variable, dependiendo de la naturaleza del juego frente al que nos encontremos. Un ejemplo sencillo que ilustra la diferencia entre jugador y rol es el siguiente: dado un juego de ajedrez convencional, se dispone de dos jugadores, A y B, y de dos roles, el jugador blanco (o ejército blanco) y el jugador negro. Supongamos que A elige ser el jugador blanco, dejando a B con las piezas negras. Arbitrariamente en el transcurso de la partida, A y B deciden intercambiar roles, esto es, A jugará con las piezas negras y B con las blancas. Esto muestra que, aunque A siga siendo A (es el mismo jugador), su rol ha cambiado durante la partida. Esta diferencia es especialmente importante de comprender porque los conceptos de rol y jugador aparecerán de manera constante a lo largo de esta memoria y se supondrá que el lector conoce la diferencia

entre los dos conceptos, ya que es algo vital para la descripción del funcionamiento del sistema cliente-servidor basado en GGP.

- **Movimientos** y **Movimientos legales** (Moves, Legal moves): un movimiento es una acción realizada por un jugador que representa un rol particular en un turno. Un movimiento legal es un movimiento que un jugador puede tomar la decisión de realizar, sin violar las reglas del juego. En cada turno, cada jugador asociado a cada rol debe seleccionar un y sólo un movimiento a realizar, y estos sólo pueden ser seleccionados de entre aquellos movimientos marcados como legales. Esto es, cada turno dispondremos de una lista asociativa de roles y movimientos seleccionados para actualizar el estado del juego. En caso de que un jugador ocupe varios roles, éste realizará las decisiones de todos sus roles, pero la lista asociativa sólo mostrará la relación rol-movimiento.
- **Estado** del juego (State): el estado del juego representa la condición particular de un juego en un momento dado, concretando más, en un turno dado. Un estado no es más que una lista asociativa que establece la relación entre las variables, definidas en la hoja de reglas, y sus valores. El estado del juego será actualizado en cada turno, tomando como entrada tanto el estado actual como los movimientos seleccionados por los jugadores.
- **Cliente** y **Sistema automático** (Client, Automatic System): aunque su significado no sea el mismo, se utilizarán ambos conceptos de manera indistinta por razones prácticas, siempre para hacer referencia a cualquier sistema conectado al servidor y con una representación existente en él.

Capítulo 2. Fundamentos tecnológicos de la solución propuesta, herramientas y metodología de trabajo

2.1 Tecnologías fundamentales

Definimos como tecnologías fundamentales aquellas que han compuesto la base del proyecto y que forman la parte más experimental de la investigación y desarrollo del prototipo del trabajo realizado.

Podemos distinguir, a grandes rasgos, tres tecnologías fundamentales aplicadas en el proyecto, que pasamos a describir con más detalle a continuación.

2.1.1 General Game Playing (GGP)

Se utiliza el término "jugadores de juegos generales" (general game players) para describir a aquellos sistemas computacionales capaces de jugar juegos de estrategia, generalmente basados en turnos, basándose únicamente en descripciones lógicas de las reglas de juego proporcionadas en tiempo de ejecución. Esto es, el sistema no conoce las reglas de juego en estado de compilación, con lo cual dichas reglas han de ser proporcionadas de manera externa.

El hecho de desconocer las reglas hasta que son proporcionadas establece una diferencia fundamental entre los jugadores de juegos generales con otros sistemas de juegos especializados, como por ejemplo Deep Blue (un computador especializado en jugar ajedrez), ya que los primeros no pueden basarse en la utilización de algoritmos especialmente diseñados para juegos específicos y, por tanto, deben descubrir ellos mismos los algoritmos a utilizar,

dependiendo así de la inteligencia del sistema y menos de los algoritmos previamente programados.

General Game Playing (GGP) define tanto la disciplina que se basa en el diseño de los sistemas antes mencionados, como un proyecto concreto de la Universidad de Stanford, California, cuya finalidad es la creación de una plataforma que consiste en un servidor que monitoriza la legalidad de los movimientos que realizan los jugadores y mantiene a los jugadores informados del estado del juego. Será este último proyecto el que consideraremos parte tecnológica del trabajo que nos ocupa, y por tanto toda referencia futura al término GGP se referirá al proyecto software de la Universidad de Stanford.

GGP, por tanto, es un proyecto software en lenguaje Java que permite la creación de sistemas capaces de procesar y aplicar reglas de juego en tiempo de ejecución, de modo que dichos sistemas sean capaces de jugar, como mencionamos previamente, sin necesidad de algoritmos específicos a diversos juegos de estrategia. GGP proporciona una librería completa para el funcionamiento de dichos sistemas, desde el procesado de las reglas de juego, la interpretación de las mismas, el flujo de los turnos y la representación de la información de cara al jugador.

Es esta librería la que utilizaremos y aplicaremos al proyecto que estamos desarrollando, ya que nos proporciona todas las herramientas necesarias para procesar reglas de actuación para sistemas automáticos y nos proporciona información clasificada para realizar una monitorización consistente y obtener conclusiones con el fin de, si procede, tomar decisiones o resolver errores.

Sin entrar en detalles sobre la estructura del prototipo, de la cual hablaremos detenidamente en el capítulo 3, merece la pena estudiar brevemente la estructura del proyecto de la Universidad de Stanford y aquellos aspectos que hacen que GGP sea una tecnología fundamental en el desarrollo de este

prototipo, así como el vocabulario y los términos que utilizaremos a lo largo de la descripción y análisis del prototipo a desarrollar:

Las clases más destacables proporcionadas por la librería GGP y utilizadas directamente en el prototipo son:

- **Game**: representa un juego, es una clase que encapsula datos sobre un juego en particular, esto es, para el caso que nos ocupa, su hoja de reglas asociada (*más en 2.1.2*). Un juego puede tener una serie de datos adicionales, como un nombre, hojas de estilo, descripción, etc., pero en nuestro caso sólo almacenaremos la hoja de reglas. Un juego que sólo almacena una hoja de reglas es llamado un Ephemeral Game (juego efímero), para distinguirlo de los juegos regulares.
- **StateMachine**: representa una máquina de estados, creada a partir de un juego (Game) inicializado con una hoja de reglas y es, probablemente, la clase más importante y más utilizada a lo largo del desarrollo del juego, ya que es la encargada de todo el procesamiento de reglas, turnos, estados y movimientos legales. Mediante diversos métodos podemos acceder al estado del juego, y actualizar dicho estado mediante selección de movimientos legales, provistos también por esta clase, con lo cual mucha parte del código del prototipo gira en torno a esta clase.
- **MachineState**: utilizada en la máquina de estados, esta clase representa, como su nombre indica, un estado de la máquina de estados (StateMachine). Esto se traduce, en términos más teóricos del proyecto, en un estado del juego. Cuando se realiza una consulta del estado actual a la máquina de estados, ésta nos devuelve un MachineState, y es este estado del juego el que, después de ser procesado, será utilizado para realizar la toma de decisiones de la forma en que el sistema cliente crea conveniente. En cada turno del juego, la máquina de estados se encuentra (como norma general) en un estado distinto, lo cual indica que cada turno del juego se

ha de realizar una actualización del estado. Aquí es donde entran en juego los movimientos legales, o Move.

- **Move**: encapsula un movimiento legal, esto es, la decisión tomada por un sistema cliente asociada a un rol definido en la hoja de reglas. Por cada rol definido, en cada turno, debe haber un Move asociado para poder realizar el procesamiento del turno y la actualización de la máquina de estados, por tanto se utilizará un HashMap (Role => List[Move]) para almacenar y organizar los movimientos legales que tiene disponible un sistema cliente para cada uno de sus roles, y el cliente deberá devolver una List con los movimientos seleccionados para poder ser comunicados a la máquina de estados y poder actualizar el estado actual del juego.

2.1.2 Game Description Language (GDL)

Aunque forma parte del proyecto GGP, y ha sido específicamente diseñado para su uso en el mismo, GDL es una parte tan primordial en el proyecto que merece ser detallada como una tecnología fundamental independiente. El Lenguaje de Descripción de Juegos, GDL a partir de ahora, es una variante de Datalog, un lenguaje de lógica declarativa que a su vez está basado en Prolog, uno de los primeros y más utilizados lenguajes de lógica de primer orden, asociado generalmente con las ramas de inteligencia artificial y lingüística computacional.

Descripción de GDL y su relación con Datalog

Uno de los objetivos de GDL y el motivo por el que fue elegido para este proyecto es por su capacidad de describir sistemas de estados sin necesidad de enumerarlos todos, ya que en sistemas mínimamente complejos esto llevaría a especificaciones muy largas, difíciles de entender y propensas a acumular errores.

La forma en la que GDL logra reducir las descripciones es mediante el uso de **flujos**. Un flujo es una condición que puede variar a lo largo del tiempo. Este concepto permite definir las funciones de transición de los sistemas basados en estados a partir de una especificación de los cambios que se producen en cada estado.

Los flujos se pueden expresar en lógica de primer orden mediante funciones o predicados con todas las variables sustituidas por constantes. Por ejemplo, en un juego es normal que en cada turno, uno o varios jugadores tengan la posibilidad de actuar sobre el estado del juego. Esto se puede expresar mediante el término función:

```
control (jugador1)
```

Para expresar, el hecho de que en el siguiente turno, es el jugador2 el que tiene la posibilidad de actuar sobre el juego podemos usar los predicados *next* y *true* del siguiente modo:

```
(<= (next (control jugador2)) (true (control jugador1)))
```

Esta cláusula expresable en GDL dice que si en el estado actual es cierto el predicado *true (control jugador1))*, entonces se puede inferir que es cierto el predicado *next (control jugador2)*, lo que supondrá que en el estado siguiente será cierto el predicado *true (control jugador2)*.

Esta es la técnica básica que usa GDL para expresar los cambios de estado: los flujos son representados por términos función con variables sustituidas por constantes, y los cambios sobre los flujos son definidos mediante cláusulas con predicados *next* y *true* que toman como argumentos estos flujos.

Por este motivo, aunque GDL se basa en Datalog, se requieren algunas modificaciones. A continuación se detallan algunas de las particularidades de GDL sobre Datalog, obtenidas a partir del documento [6]:

Para empezar se amplía el vocabulario de constantes: a las constantes que representan las relaciones y los objetos constantes, se añaden las constantes que representan funciones. Además, los términos podrán ser ahora: variables, objetos constantes y se añaden funciones de cierta aridad.

La regla de Datalog mantiene su estructura: $h \Leftarrow b_1 \wedge b_2 \ldots \wedge b_n$, siendo h una sentencia atómica que da lugar a la cabeza de la regla y el cuerpo está formado por una conjunción de literales. Ahora bien, en GDL cada regla debe respetar una restricción de recursión.

Esta restricción viene del hecho de que en Datalog con funciones y recursión (ciclos), la resolución de una consulta no es decidible, e incluso puede llevar a múltiples respuestas. Puesto que es necesario para que el sistema sea operativo que se puedan resolver consultas (por ejemplo, para saber qué predicados next serán ciertos tras cada estado), hay que resolver este problema. GDL lo consigue imponiendo lo siguiente:

Sea la regla $p(t_1, t_2, \ldots, t_n) \Leftarrow b_1 \ldots \wedge q(v_1, \ldots, v_k) \ldots \wedge b_m$ perteneciente a un conjunto de reglas. Supongamos que p y q están dentro de un mismo ciclo en el grafo de dependencias del anterior conjunto de reglas. Entonces, para cada argumento v_i de q, se cumple que:

- Es una constante.
- O bien, coincide con uno de los argumentos de p.
- O bien, alguno de los restantes argumentos del cuerpo de la regla, es un predicado que no está en ningún ciclo con q y alguno de sus argumentos es igual al argumento v_l.

Con esta restricción evitamos que al tratar de resolver la consulta, los términos función empiecen a crecer al entrar recursivamente en los ciclos.

La última diferencia con Datalog es que GDL no incluye la relación de igualdad entre términos, como por ejemplo X=Y. La razón es que si se quiere incluir la comprobación de esta relación como parte del cuerpo de una regla, se puede sustituir uno de los términos en un lado de la igualdad por el otro en la propia regla. Los que sí define el lenguaje GDL es el predicado distinct, para comprobar si dos términos son diferentes.

Especificación básica de predicados predefinidos

GDL establece un conjunto de predicados predefinidos:

- El predicado *role*. Establece los objetos constante que hacen las veces de agentes en el juego.
- Los predicados *next*, *true* e *init*. Hay que tener en cuenta que en GDL el estado se representa por flujos definidos como términos función "*grounded*". El predicado *true* permite definir cuáles de estos términos función son parte del estado actual, mientras el predicado *init* permite definir cuáles son parte del estado inicial. Por su parte, el predicado *next* establece qué flujos son parte del siguiente estado.
- El predicado *legal*. Esta relación tiene dos argumentos: el primero debe ser una constante para la que el predicado *role* se cumple. El segundo es una acción del agente, que bajo ciertas condiciones podrá ser posible y para otras no, por tanto se trata de un término función. Lo siguiente es un ejemplo de este predicado: `(<= (legal P (mark X Y)) (and (true (cell (X, Y, b)) (true (control (P)))))`
- El predicado *does*. Sirve para representar la elección de una acción por parte de un agente. Tiene dos argumentos: una constante para la que el

predicado *role* es cierto y un término función que es una acción. Se utilizará en las cláusulas *next* para indicar el cambio de estado.

- Los predicados *goal* y *terminal*. El predicado *terminal* sirve para especificar el estado final del juego, mientras que el predicado *goal* establece la función de utilidad.

Además, para mantener la semántica de los predicados anteriores, GDL establece un conjunto de restricciones sobre la utilización de los mismos:

- El predicado *role* sólo puede aparecer en sentencias atómicas "grounded".
- El predicado *init* sólo puede aparecer en la cabecera de las cláusulas y no puede aparecer en el grafo de dependencias conectado a predicados *true*, *does*, *next*, *legal*, *goal* y *terminal*.
- El predicado *true* no puede aparecer en la cabecera de las cláusulas.
- El predicado *next* sólo puede aparecer en la cabecera de las cláusulas.
- El predicado *legal* sólo puede aparecer en el cuerpo de las cláusulas y en el grafo de dependencias no puede estar conectado con *does*, *goal* o *terminal*.

Especificación de las hojas de reglas

Como ya se ha mencionado, GDL fue diseñado y desarrollado específicamente para su utilización con el proyecto GGP. Los jugadores de juegos generales, al no conocer las reglas del juego ni ningún algoritmo específico para la toma de decisiones durante los mismos, necesitan recibir de manera externa una serie de pautas que delimiten sus posibilidades a la hora de realizar acciones en un juego. Necesitan reglas de juego.

GDL define las reglas por las que se rige un juego, o en nuestro caso, un MDSS, mediante lo que llamaremos *hoja de reglas* (o *rulesheet*). Las hojas de reglas son ficheros que contienen una serie de predicados que representan las

mecánicas de juego como reglas lógicas y el estado del juego en términos de un conjunto de hechos verdaderos [6].

Una hoja de reglas está conformada por:

- **Roles**: los roles representan las distintas partes que realizan acciones en un juego o, en nuestro caso, en un sistema automático. Por ejemplo, el jugador blanco en ajedrez representa un rol, y puede ser expresado en GDL como (*role white_player*). En nuestro caso, un sensor de luz podría ser un rol, ya que es una entidad que realiza acciones y manipula información durante el desarrollo del juego.
- **Inicialización de valores y acciones**: lo primero que debemos hacer a la hora de redactar una hoja de reglas, es definir cuál será el estado inicial del juego, para poder trabajar con un punto de partida concreto, las acciones que pueden realizar los roles y los valores que pueden tomar distintas variables definidas de cara al funcionamiento de dichas acciones. Siguiendo el ejemplo del juego de ajedrez, podemos definir algunos valores iniciales de dicho juego (no se van a contemplar todos, ya que es un ejemplo sencillo) de la siguiente manera:

Elemento	Ejemplo	Explicación
Tipos de valor	`(Tilevalue black)` `(Tilevalue white)`	Representa los dos posibles valores que puede tomar una variable del tipo Tilevalue. En este caso, por ejemplo, las casillas pueden ser negras o pueden ser blancas.
Base	`(<= (base (tile ?x ?y bishop)) (Index ?x) (Index ?y))`	La base determina restricciones sobre los valores de los elementos como, en este caso *tile*, que representa las casillas del tablero. En este caso, se está indicando en la hoja de reglas, que una casilla de cualquier índice (x, y) puede contener un alfil (*bishop*).
Input	`(<= (input ?p (move ?p ?x ?y)) (Piece ?p) (Index ?x)`	En este caso, definimos una acción que puede ser llevada a cabo por un rol. En el ejemplo, estamos

	`(Index ?y) (Role ?r))`	indicando que una pieza cualquiera, del tipo Piece, puede moverse a cualquier casilla (x, y) que el jugador p le indique, y que esta acción puede ser realizada por cualquier rol. Una pieza tiene la capacidad de moverse a cualquier casilla, pero no estamos contemplando la legalidad de los movimientos en esta fase ya que esto se definirá luego.
Inicialización	`(init (tile 1 a rook))`	Indicamos que el elemento que se encuentra en la casilla $(1, a)$ en el inicio del juego es una torre.

Tabla 2.1 Secciones y elementos de las hojas de reglas

- Reglas que determinan los **movimientos legales** de cada rol: como vimos en el ejemplo anterior, con el predicado *input* se definen los movimientos que pueden realizar los distintos roles, pero esto no garantiza que el movimiento descrito sea legal en una instancia particular del estado del juego. Por ello hay que definir las reglas en las que se basa el conjunto de acciones que pueden tomar, de manera legal, los distintos roles que participan en el juego, mediante el predicado *legal*. Siguiendo con el ejemplo y para introducir el predicado *noop*, podemos estudiar el siguiente predicado simple: *(<= (legal white_player noop) (true (control black_player)))*. Este predicado establece que es legal que el jugador blanco realice la acción *noop* siempre y cuando sea verdadero que el control del juego (el turno) pertenece al jugador negro. La acción noop indica, por convenio, que el jugador blanco no realiza ninguna acción ese turno. Dada la naturaleza del ajedrez (y de muchos otros juegos de mesa), esta es, generalmente, la única acción legal que puede realizar el jugador que no controla el turno, mientras que el jugador que lo controla, en este caso el jugador negro, dispondrá de movimientos legales para manipular directamente el estado del juego, como mover piezas.
- Reglas que determinan el **estado futuro** basado en los movimientos que se realicen: mediante el predicado *next*, se puede establecer una serie de

reglas que se utilizarán para determinar el estado del juego en el turno siguiente, dependiendo del estado actual y, en algunos casos, de las acciones que el jugador haya decidido llevar a cabo, incluyendo el rol que tendrá el control el siguiente turno. Por ejemplo, *(<= (next (Control white_player)) (true (Control black_player)))* indica que el jugador blanco será el próximo controlador del turno, siempre y cuando sea cierto que el control esté asignado actualmente al jugador negro. Esto significa, en pocas palabras, que el predicado *next* establece la función de transición de la máquina de estados. Cabe destacar que el predicado *next* también se ha de utilizar de manera explícita para guardar el estado de los elementos del juego no manipulados, es decir, aquellas variables que no hayan sido parte de alguna acción de algún rol este turno, actuando a modo de memoria, ya que hay que considerar que en la representación de estado de GGP y, por extensión, también de este prototipo, se asume en todo momento el convenio de "*mundo cerrado*" ("*closed world*"), de modo que todo predicado que no se encuentre explícitamente representado en el estado actual o en la hoja de reglas, se considera falso.

2.1.3 Redes Lógicas de Markov y Alchemy 2.0

Destinado como base para la monitorización de posibles fallos en los sistemas automáticos, utilizaremos el concepto de Redes Lógicas de Markov (Markov Logic Networks, o MLN). Una Red Lógica de Markov se puede definir, tomando prestada la definición en [7], como una base de conocimiento de primer orden, en la que cada fórmula o predicado tiene un peso asociado. Asimismo, desde el punto de vista de la lógica de primer orden, las MLN proporcionan la habilidad para manejar conceptos de incertidumbre, tolerar imperfecciones y conocimiento contradictorio.

Para ampliar la información al respecto de las MLN, utilizaremos la introducción extraída de [8].

La semántica de una red lógica de Markov, se fundamenta en el modelo gráfico de distribución de probabilidad conocido como red de Markov. En una red de Markov, tenemos un conjunto de variables aleatorias asociadas a los nodos de un grafo no dirigido. Este grafo representa una distribución de probabilidad para el conjunto de variables X que viene dada por la fórmula:

$$P(X = x) = \frac{1}{Z}\prod_{k}\varphi_k(x_{\{k\}})$$

donde x es el valor del conjunto de variables, y las funciones φ_k se denominan potenciales y deben ser funciones con valor positivo y estar asociadas a los cliques del grafo. Así, $x_{\{k\}}$ es el valor de las variables incluidas en el k-ésimo clique del grafo. Finalmente, Z es una función de partición que normaliza la suma de las probabilidades para todas las valoraciones de las variables X a 1.

Una forma equivalente de expresar este modelo de distribución de probabilidad es:

$$P(X = x) = \frac{1}{Z}\exp(\sum_{j}\omega_j f_j(x))$$

donde $f_j(x)$ se denomina "característica". La transformación de una versión a otra, supone reemplazar cada función potencial por una exponencial de una suma de características del estado, cada una con un peso ω_j.

Una manera posible de hacer esta transformación consiste en establecer una característica para cada estado posible de cada clique. Supongamos entonces la característica asociada al clique k-ésimo cuando el estado es $x\{k\}$, y denotémosla por $f_{x\{k\}}(x)$. Entonces, esta característica será 1 cuando el valor

de las variables X incluya a $x\{k\}$ y 0 en caso contrario. Con estas características definidas así, el peso asociado a $f_{x\{k\}}(x)$ será $\log \varphi_k(x_{\{k\}})$ para obtener el modelo original en términos de las funciones potencial del clique.

Una de las ventajas de este modelo es que la inferencia es #P-completa, debido a una propiedad importante de las redes de Markov:

$$\forall X_j \notin MB(X_i), P\big(X_i = x_i \big| MB(X_i), X_j\big) = P(X_i = x_i | MB(X_i))$$

donde $MB(X_i)$ representa las variables que son vecinas de la variable X_i en el grafo (se denomina Markov blanket). Estas variables suponen un "aislamiento" de la variable respecto al resto de variables, ya que como vemos, dado un conjunto de valores establecidos para las variables del Markov blanket, la variable X_i es condicionalmente independiente de cualquier otra variable del conjunto. Gracias a esta propiedad de las distribuciones de probabilidad asimilables a modelos de red de Markov, se pueden realizar inferencias aproximadas (cálculo de probabilidades condicionadas y marginales) de forma relativamente eficiente.

A partir de estos modelos podemos definir lo que se entiende por una red lógica de Markov o MLN. Se trata de un conjunto L de pares (F_i, w_i), donde F_i es una fórmula en lógica de primer orden y w_i es un número real. De modo que si a L le añadimos un conjunto finito de constantes, vamos a poder definir una red de Markov $M_{L,C}$ del siguiente modo:

1. $M_{L,C}$ contiene una variable binaria para representar cada posible sustitución de variables por constantes en cada predicado de L. El valor de esta variable es 1 si el predicado es verdadero una vez hecha la sustitución y 0 en caso contrario.
2. $M_{L,C}$ contiene una característica para cada posible sustitución en una fórmula F_i de L. El valor de esta característica es 1, si la fórmula es

verdadera una vez aplicada la sustitución y 0 en caso contrario. El peso de la característica es el peso w_i asociado con la fórmula en L.

El proceso que nos está diciendo esta definición, consiste en tomar un conjunto de fórmulas de primer orden y realizar todas las sustituciones de variables posibles para tener una familia de fórmulas básicas. A continuación crearemos un grafo donde cada nodo será un predicado básico diferente extraído de las fórmulas básicas, y añadiremos arcos al grafo uniendo entre sí, todos aquellos predicados que compartan fórmula básica. Este grafo se denomina "red de Markov básica" (grounded Markov Network). Podemos ver que cada fórmula básica debe corresponderse con un clique del grafo. El peso asociado a cada fórmula básica es el w_i de la fórmula generadora en lógica de primer orden. Así, si el número de fórmulas básicas verdaderas resultantes de las sustituciones de variables en una fórmula con peso w_i es $n_i(x)$, el modelo de red de Markov establece que para esa sustitución de variables:

$$P(X = x) = \frac{1}{Z}\exp(\sum_j w_i n_i\,(x))$$

Repasemos ahora el concepto de "mundo posible" en términos de la lógica de primer orden. Un mundo posible es un conjunto de constantes, funciones, un conjunto de relaciones entre estos objetos y una asignación de verdad a cada una de estas relaciones (interpretación). Si vemos como se construye la red de Markov, $M_{L,C}$, conteniendo todos los predicados base que se generan a partir de las fórmulas lógicas, vemos que un estado de dicha red (asignación de 1s y 0s a los diferentes nodos) no es más que la especificación de un "mundo posible".

Bajo un conjunto de suposiciones adicionales, que son innecesarias cuando el número de constantes es finito (consultar [8]), la red de Markov $M_{L,C}$ es una

distribución de probabilidad de estos mundos posibles. No olvidemos que en este caso la asignación de valores al conjunto de variables X representa la asignación de 1 o 0 a cada predicado básico de la red.

Esta distribución de probabilidad tiene una característica intuitiva sobre el significado de los pesos que asociamos a las fórmulas lógicas. Un peso positivo elevado en relación al resto de pesos, para una fórmula indica que los "mundos posibles" donde esa fórmula sea falsa tendrán asociada una probabilidad pequeña, mientras que si el peso tiende a ser 0, esa fórmula será irrelevante a la hora de determinar la probabilidad de los "mundos posibles" que cumplan o dejen de cumplir la fórmula.

En este sentido podemos ver la red lógica de Markov como una gradación de los valores de verdad de las bases de fórmulas en lógica de primer orden. En la semántica de la lógica de primer orden, la base de fórmulas es verdadera bajo una interpretación si las fórmulas que componen la base son verdaderas bajo las reglas de esta semántica y el valor de verdad de los predicados de la interpretación. En el caso de la red lógica de Markov, no tenemos una decisión de tipo verdadero/falso para la base de reglas, sino que tenemos una distribución de probabilidad para las diferentes interpretaciones. Aquí estableceremos probabilidades para las diferentes fórmulas que conforman la red lógica de Markov.

Así, la tarea de inferencia consiste en calcular $P(F_1 \mid F_2, M_{L,C})$, es decir, la probabilidad de la fórmula F_1 (que denominamos consulta o query), dada la fórmula F_2 que denominamos evidencia, asumiendo la distribución de probabilidad para los mundos posibles determinada por la red de Markov $M_{L,C}$.

Si aplicamos la regla de Bayes, obtenemos:

$$\mathrm{P}(F_1|F_2, M_{L,C}) = \frac{P(F_1 \wedge F_2|M_{L,C})}{P(F_2|M_{L,C})}$$

Pero, por una parte:

$$P\big(F_2 \wedge F_1\big|M_{L,C}\big) = \sum_{x \in \chi_{F_1} \cap \chi_{F_2}} P(X = x|M_{L,C})$$

donde χ_{F_i} son los mundos posibles donde la fórmula F_i es cierta, y por otra :

$$P\big(F_2\big|M_{L,C}\big) = \sum_{x \in \chi_{F_2}} P(X = x|M_{L,C})$$

Por lo tanto, resulta que el problema de la inferencia se reduce a un cálculo de probabilidades de mundos posibles:

$$P\big(F_1\big|F_2, M_{L,C}\big) = \frac{\sum_{x \in \chi_{F_1} \cap \chi_{F_2}} P(X = x|M_{L,C})}{\sum_{x \in \chi_{F_2}} P(X = x|M_{L,C})}$$

Este cálculo involucra la inferencia lógica, que es NP-completo, con lo que en general el problema de inferencia sería intratable. Sin embargo, existen algoritmos de inferencia aproximada que permiten calcular estimaciones aceptables.

El objetivo de esta introducción a las redes lógicas de Markov, ha sido justificar su uso en el problema de la monitorización y llevar a una comprensión del significado de los pesos en la base de reglas. Si una fórmula de la red lógica de Markov tiene un peso elevado que lleva a hacer poco probables los mundos posibles donde una fórmula es cierta, esta última fórmula será poco probable. En el caso de que se condicione la probabilidad

a una evidencia, esto hay que matizarlo, ya que si los mundos posibles donde la evidencia es cierta también son poco probables, la probabilidad condicionada puede no ser tan pequeña como se aprecia en la última fórmula (la probabilidad de los mundos posibles de la evidencia aparece en el divisor).

Nos encontramos pues ante un concepto aplicable a la hora de tratar con situaciones que, dadas las reglas y las mecánicas de un juego concreto, son posibles, pero que no tienen sentido o son incoherentes para los sistemas de toma de decisiones, que son generalmente los sistemas cliente, aquellos que entienden y/o interpretan la información obtenida desde el servidor de juegos generales para intentar obtener conclusiones.

Como tecnología concreta para el uso de este concepto, se hará uso de Alchemy 2.0. Éste es un paquete software para inferencia y aprendizaje en Redes Lógicas de Markov [9], desarrollado en la Universidad de Washington y que nos proporciona la capacidad de inferir, dada una MLN y una serie de evidencias observadas, la probabilidad de que uno o más predicados sean verdaderos, lo cual lo hace bastante adecuado, al menos como concepto experimental, para intentar abordar el tema de la monitorización de fallos.

Un fichero MLN adecuado contiene, además de especificaciones de tipos y variables, un conjunto de fórmulas lógicas, o predicados, con un peso asignado para cada una. El valor asignado a cada predicado es arbitrario y su valor ponderado será dependiente del resto de valores especificados en el fichero. Por ejemplo:

```
1.0 Lightvalue(t,LOff)^Bulbvalue(t,BOn)=>Error(SensorOff)

100.0 Lightvalue(t,LOn)=>!Error(SensorOff)
```

El primer predicado indica que si *Lightvalue* se encuentra en el valor *LOff* y *Bulbvalue* en *BOn*, esto implica que *Error(SensorOff)* es verdadero con un peso asignado de 1.0. En términos más coloquiales, si sabemos, como hecho, que una bombilla se encuentra encendida y el sensor de luz, que debería estar detectando algún nivel de luminosidad, no lo está, es probable que exista un error y que, en este caso particular, probablemente sea del tipo *SensorOff* (el sensor no detecta luminosidad en cualquier condición dada). Hablamos en términos de probabilidad ya que no es 100% seguro que sea esto lo que está sucediendo. Por este motivo se asigna un peso a cada predicado lógico en el fichero MLN.

El segundo predicado, con un peso asignado considerablemente mayor, de 100.0, simplemente indica que si el sensor detecta algún tipo de fuente de luz, causando que éste se encuentre en el estado *LOn*, no puede ser cierto que haya un fallo del sensor.

Si se toman estos y otros predicados y se aplican a una serie de evidencias, en una base de conocimiento, se puede inferir si existe algún tipo de fallo y con qué probabilidad puede existir.

2.2 Tecnologías de apoyo

Como tecnologías de apoyo, se describirán aquellas herramientas utilizadas a la hora de realizar el desarrollo del prototipo, pero que no forman parte de la base fundamental teórica del proyecto.

2.2.1 Scala

Para construir las estructuras sobre las que se apoyarán las tecnologías fundamentales del prototipo, se ha decidido utilizar Scala, un lenguaje que fusiona la ya clásica programación orientada a objetos con las características

más utilizadas y relevantes de los lenguajes de programación funcional, como Haskell o, indirectamente, Lisp. Para más información extendida sobre el lenguaje Scala, que no se considera pertinente discutir en esta memoria, consultar [10].

Figura 2.1 Logotipo de Scala

Scala es un lenguaje compilado y fuertemente tipado, aunque con inferencia de tipos, cuyos programas se ejecutan sobre la Máquina Virtual de Java (JVM). Para poder utilizar la JVM, el compilador produce una salida en Java Bytecodes, el conjunto de instrucciones de la JVM, que es el mismo tipo de salida que produce el compilador de Java (y otros lenguajes como Clojure, por ejemplo). Esta característica particular de Scala lo hace completamente compatible con código Java, lo cual lo hace una elección razonable para el desarrollo del proyecto que nos ocupa, ya que las librerías de GGP están desarrolladas en lenguaje Java y era necesario un lenguaje que pudiese utilizar las herramientas proporcionadas por el mismo, preferiblemente, de manera directa y sin barreras.

Además de la completa compatibilidad con el código Java, otra motivación que llevó al desarrollo del prototipo en lenguaje Scala fue, como ya se ha mencionado antes, sus características funcionales. Una de las características fundamentales que se quería aprovechar era la noción de funciones lambda (o funciones anónimas), que nos permite pasar funciones como parámetros a otras funciones u objetos, ya que en este paradigma, las funciones son consideradas objetos. Esto quiere decir que se puede asignar un bloque de

código a una variable o constante y trabajar con ella como un objeto. Lo interesante de todo esto radica en que facilita de manera considerable la manipulación de colecciones de datos (en nuestro caso concreto, mayoritariamente hashes) siguiendo una naturaleza inmutable, de modo que se evite en la medida de lo posible incoherencias en la información transmitida y almacenada.

Por último, otro de los motivos por los que se eligió Scala como lenguaje de desarrollo fue la utilización de la librería de actores Akka, que describiremos en la siguiente sección.

Nota: Desde la versión 8 de Java, este lenguaje también dispone de funciones lambda, lo cual lo hace perfectamente viable para el desarrollo de este prototipo, pero a la fecha en la cual se empezó a programar el prototipo del proyecto, no se encontraba disponible dicha versión de Java y, dada la naturaleza experimental del proyecto, se eligió la utilización de Scala frente a Java en aras de experimentar todo lo posible a nivel de tecnologías y estructuración del proyecto, además de la ya mencionada facilidad de manipulación de estructuras que nos ofrecía Scala y sus funciones lambda.

2.2.2 Akka

Como se mencionó brevemente en la sección anterior, se ha utilizado una librería de actores llamada Akka. Akka está definida en su sitio web principal [11] como un conjunto de herramientas y un entorno de ejecución para construir aplicaciones altamente concurrentes, distribuidas y tolerantes a fallos, orientadas a la JVM.

Akka puede ser utilizada tanto con Scala como con Java y consiste, centrándonos en el proyecto que nos ocupa, en una librería que nos proporciona entidades ligeras llamadas actores. Un actor es una clase, un

contenedor, que contiene varios elementos fundamentales: estado, comportamiento, un buzón de mensajes, actores hijos y una estrategia de supervisión de los anteriores. Todo esto está encapsulado tras el concepto de referencia de actores [12].

Figura 2.2 Logotipo de Akka

Para no saturar al lector con información que no es completamente relevante, nos limitaremos a hablar de las características más importantes y más utilizadas a lo largo del desarrollo del proyecto.

Como ya hemos mencionado, un actor es una clase que, en el caso de Akka, extiende el *trait* Actor. Un trait es un concepto similar (para el caso que nos ocupa, igual) al de interfaces de Java. Por lo tanto una clase que quiera utilizar las características de un actor, debe extender, implementar, los métodos que proporciona el *trait*. Sólamente existe un método obligatorio a la hora de implementar Actor: el método *receive*.

Un actor encapsula, citando parcialmente lo anteriormente comentado, comportamiento y un buzón de mensajes, además de otros elementos. La característica fundamental que distingue un actor de una clase convencional es que el estado no es accedido o manipulado directamente de forma externa, y el comportamiento, de igual modo, no es modificado o consultado mediante llamadas a métodos, como se hace habitualmente. Un actor puede contener todos los datos que se deseen, como una clase convencional, mediante

atributos, pero es sobre todo el comportamiento lo que difiere en el modelo de actores. Los actores se comunican mediante una interfaz de mensajes, mediante la implementación del método receive. Este método determina el comportamiento que seguirá un actor al recibir mensajes de distintos tipos, y es la única manera adaptada al modelo de actores que permite acceder a los datos del mismo.

Por poner un ejemplo sencillo:

```
class SampleActor extends Actor {
   def receive = {
       case Hello    =>  sender ! Hi
       case Goodbye  =>  sender ! Bye
       case _        =>  sender ! "I can't understand you"
   }
}
```

En este caso, esta sencilla implementación de un actor de ejemplo, *SampleActor*, establece el comportamiento siguiente: cuando nuestro actor de ejemplo recibe un mensaje de la clase *Hello* o Goodbye, devuelve una contestación de la clase *Hi* o *Bye*, respectivamente. Podemos ver que los mensajes son clases (específicamente, *case classes*) que pueden tener atributos, constructores, o cualquier cosa que se necesite. Por último, el caso por defecto establece que nuestro actor, cuando reciba algo que no corresponda a las clases para las que está programado, envía un mensaje, de clase String, que encapsula el texto "*I can't understand you*".

Cuando un actor recibe un mensaje, éste se almacena en el buzón de mensajes a la espera de ser procesado. Cuando el actor está listo para procesar el mensaje, el método *receive* pasa a la acción, aplicando la acción correspondiente dependiendo el tipo de mensaje que vaya a procesar.

Lo interesante, y el motivo por el cual se eligió utilizar un modelo de actores en el proyecto, es que permite implementar concurrencia en el prototipo de manera relativamente sencilla. El modo de procesamiento de los mensajes hace que una aplicación que basa sus cimientos en Akka (o cualquier otra implementación del modelo de actores) tenga una naturaleza asíncrona por defecto, permitiendo en este caso la comunicación con gran cantidad de sistemas automáticos a la vez sin necesidad de programar toda la estructura necesaria para implementar comunicación asíncrona, y dando prioridad a la programación del comportamiento de los actores, que son los que cargan la lógica de negocio de la aplicación.

2.2.3 Spray

Para realizar la comunicación entre el servidor y los distintos sistemas automáticos (o clientes) que se conectan al mismo, como ya hemos mencionado en la sección de solución propuesta, se ha utilizado una interfaz REST (una interfaz basada en peticiones HTTP).

Figura 2.3 Logotipo de Spray

Sin entrar demasiado en detalles, porque esta tecnología no es tan relevante, pero sin embargo merece ser mencionada, Spray [13] es una librería de comunicación HTTP basada en Akka. Esto quiere decir que el servidor HTTP es un sistema de actores, y el comportamiento del mismo de cara a las peticiones será programado en el método receive del actor principal del sistema, con lo cual desde el primer contacto con el servidor prototipo

desarrollado en el proyecto, estaremos tratando con una arquitectura basada en actores.

2.3 Herramientas y metodología de trabajo

Para el desarrollo del prototipo se ha intentado utilizar herramientas basadas en software libre que nos permitan trabajar de forma cómoda y eficiente. Por este motivo, el entorno de trabajo que se ha utilizado es una máquina **Linux** (concretamente se han utilizado varias máquinas con Xubuntu, Fedora y Mageia como sistemas operativos, de modo que se compruebe que el prototipo funcione en distintos entornos que, aunque todos sean Linux, tienen ligeros cambios en su funcionamiento con respecto a ciertas tecnologías). Algunas de estas máquinas han sido de 64 bits, otras de 32, con lo cual se ha comprobado que la aplicación funciona en distintos entornos, como era de esperar dada la flexibilidad de las tecnologías utilizadas.

Además del sistema operativo, se ha utilizado el entorno de desarrollo (IDE) Eclipse, en una variante llamada **Scala IDE** [14], específicamente diseñado para, como su nombre indica, su uso específico con el lenguaje Scala. La versión de Eclipse en la que se basa el entorno utilizado es 4.3 (Kepler).

Para facilitar la automatización de tareas se ha utilizado la herramienta ***sbt*** [15], que nos proporciona diversas ventajas y comodidades a la hora de trabajar con tareas repetitivas como la depuración manual de una arquitectura cliente-servidor. Es una herramienta similar a Apache Maven, que unifica la posibilidad de realizar tareas que afectan a todo el proyecto como compilar, ejecutar, gestionar dependencias, metainformación sobre el proyecto, etc.

Entrando un poco en la metodología utilizada durante el desarrollo del prototipo, se ha optado por una metodología centrada en el diseño y marcada

por los objetivos acordados para este trabajo. Esto es, se ha puesto gran importancia en cómo realizar un diseño para que el prototipo cumpla sus funciones y sea, a su vez, lo más sencillo posible de comprender para posibles programadores futuros de modificaciones del mismo, intentando elegir una estructura basada en capas (que discutiremos en el siguiente capítulo) y con realización de pruebas, sobre todo manuales, gracias a la automatización de algunos procesos por parte de *sbt*.

Adicionalmente, para el seguimiento de las tareas a realizar durante el desarrollo se han utilizado tickets de **GitHub** [16], con lo cual esta conocida plataforma ha actuado a la vez como repositorio convencional de software y como gestor para el seguimiento de las tareas pendientes.

Capítulo 3. Descripción y características del prototipo

En este capítulo, se procederá a desglosar la estructura del prototipo propuesto, brevemente descrito anteriormente en esta memoria. Para la descripción del prototipo en su totalidad se adoptará un enfoque top-down, de modo que primero se expliquen y describan los elementos de manera más general para obtener una visión de conjunto y luego se proceda a detallar cada una de las piezas que conforman dichos elementos, para una visión más particular.

3.1 Visión general

3.1.1 Descripción básica y justificación de la arquitectura general

Nos encontramos ante un prototipo basado en una estructura clásica de cliente-servidor, en que los clientes deben conectarse a dicho servidor de manera proactiva para solicitar servicios del mismo mediante una interfaz REST (HTTP). Dada la naturaleza y la temática de este Trabajo de Fin de Grado, y los objetivos propuestos (ya descritos en el capítulo 1), la mayor parte de nuestros esfuerzos serán orientados al desarrollo del lado del servidor, con lo cual, gran parte de los elementos que se discutirán en este capítulo serán parte del servidor, mientras que el cliente desarrollado será tratado como una referencia a cualquier tipo de cliente general con la capacidad de conectarse y utilizar las características y servicios que proporciona el servidor.

Se ha seleccionado esta arquitectura con la finalidad de satisfacer las necesidades de monitorización y control de toma de decisiones de diversos clientes al mismo tiempo, de manera centralizada e intentando reducir la carga de procesamiento de datos en los clientes, los cuales tienen como

características deseables, de cara a IoT, la ligereza, la velocidad con la que realizan las tareas específicas que representan su finalidad en la red y un bajo consumo de energía. Trasladando la gran mayoría de la carga de monitorización y análisis de datos extraídos al servidor, podemos intentar aproximarnos un poco más a estas características ideales, además de permitirnos utilizar herramientas específicas de aprendizaje e inferencia en aras de analizar la información que nos proporcionan los sistemas automáticos conectados al servidor sin tener que depender de la homogeneidad de los clientes más allá de la API.

3.1.2 Responsabilidades de las partes implicadas

Partiendo de la arquitectura de cliente-servidor, debemos dejar claras las responsabilidades que son asignadas a cada una de estas dos partes que engloban todos los conceptos que describiremos más adelante en este capítulo.

Cliente

Un cliente, como ya hemos comentado en la sección de conceptos, es todo aquel sistema capaz de realizar una conexión con el servidor y que tenga una representación virtual en el mismo después de una inicialización. Del propio concepto genérico de cliente y de esta definición aplicada al proyecto, podemos deducir que un cliente tiene las siguientes responsabilidades:

- Conectarse de manera proactiva o, más concretamente, ser capaz de generar peticiones HTTP hacia el servidor. Para ello se utiliza generalmente una API de peticiones HTTP. En el caso de nuestro cliente prueba, utilizamos un cliente HTTP bastante básico, utilizando Spray y Scala, para generar las peticiones necesarias para el correcto flujo de la información entre el cliente y el servidor. A pesar de que, en nuestro caso utilizamos las tecnologías mencionadas anteriormente, cualquier API, en

el lenguaje que se prefiera, que sea capaz de generar peticiones HTTP y recibir las respuestas pertinentes será perfectamente viable para realizar las conexiones al servidor prototipo. Gracias a esto, podemos garantizar la independencia lingüística y tecnológica de los clientes, y podremos tratar con sistemas automáticos heterogéneos siempre y cuando el resto de las responsabilidades de los mismos se cumplan.

- Al resolver correctamente una petición HTTP un cliente recibirá una serie de datos estructurados con una finalidad específica. Serán los clientes los responsables de procesar, de la manera que se elija, dicha información. Esto incluye el ID asignado a cada cliente, el estado del juego para tomar decisiones, los movimientos legales en cada turno, etc. Para facilitar el procesamiento de estos datos y garantizar la universalidad de los mismos, las colecciones que sean enviadas desde y hacia el servidor, serán siempre convertidas a formato JSON, considerado un estándar en la transmisión de objetos basados en clave-valor.
- Los clientes serán aquellos que proporcionen al servidor tanto la hoja de reglas en lenguaje GDL como el fichero MLN que contenga los predicados y sus correspondientes pesos. Esto quiere decir que los clientes aportarán al servidor toda aquella información específica sobre su modo de funcionamiento de cara a GGP y Alchemy, ya que el servidor no tiene por qué conocer ningún detalle concreto sobre el sistema que representa cada cliente de antemano.
- Serán también los clientes, siguiendo la lógica anterior, los responsables de tomar todas las decisiones a la hora de seleccionar los movimientos a realizar en cada turno, ya que el servidor no es capaz de generar el conocimiento necesario, aunque posea la información, para tomar dichas decisiones. Será por tanto el cliente el que decidirá, siguiendo el algoritmo que más le convenga, qué movimientos debe realizar cada rol en cada turno, acotados por el conjunto de movimientos legales de los que se dispone. Dado que el cliente será el que tome las decisiones de cada rol, se

considerará, en este prototipo, que el cliente es el único jugador de su juego, generando así, en la práctica, un juego con un sólo jugador que dirige varios roles.

Servidor

El servidor representa el núcleo del prototipo y es el encargado de realizar la mayor parte del trabajo que permite que las peticiones HTTP de los clientes sean correctamente resueltas, siempre y cuando el cliente proporcione la información necesaria para resolver dichas peticiones. Sus responsabilidades se comentan a continuación:

- El servidor será el encargado de recibir y procesar todas las peticiones HTTP que los clientes envíen, devolviendo la información solicitada si se procesa correctamente, o devolviendo un error en caso contrario. Esto incluye, considerando las capacidades actuales del prototipo, inicializar un juego y asignarlo a un cliente, informar sobre el estado del juego, informar sobre los movimientos legales de los que disponen los roles de un juego y actualizar el estado de los mismos, utilizando información tanto interna como proporcionada por el cliente.
- El servidor está compuesto mayoritariamente por una jerarquía de actores que son las entidades que procesan el trabajo, lo organizan y lo distribuyen. Será el propio servidor, evidentemente, el que se deberá garantizar que la comunicación entre los actores internos y los modelos de representación, así como de la capa HTTP, es correcta y se transmite la información necesaria para realizar el procesamiento de la información.
- El servidor contendrá modelos representativos de cada sistema automático conectado a él y correctamente inicializado. Será parte de la responsabilidad del servidor asegurarse de que cada modelo tiene un juego efímero asignado, inicializado a partir de la hoja de reglas que se reciba del

cliente. A su vez, el servidor asignará a cada cliente un ID para las futuras conexiones que se realicen y estén relacionadas con el mismo juego (cliente).

- Por último, el servidor será el encargado de almacenar toda la información pertinente para la realización de tareas de inferencia e interpretación de datos. Esto es, el servidor mantendrá un histórico de los estados por los que ha pasado cada juego representado en el servidor.

3.2 Arquitectura del sistema desde el punto de vista funcional

La estructura planteada permite el siguiente esquema funcional a la hora de automatizar un sistema.

El problema de automatización complejo, con múltiples objetivos y elementos (sensores, actuadores, controladores) podría dividirse en problemas más simples. Cada uno de estos subproblemas equivaldría a un cliente en el sistema propuesto. Dichos clientes representarían el subproblema o entorno multiagente mediante una descripción GDL y una MLN para la monitorización. El cliente utilizaría el servidor propuesto para obtener tanto información sobre las operaciones que el controlador puede seleccionar para los actuadores, como información de monitorización.

Podemos plantear un ejemplo con un sensor RFID, el actuador sobre el punto de luz, la etiqueta RFID, un reloj y un sistema automático que controla la persiana de la ventana. La idea sería que el sensor RFID tendría un rol diferente dependiendo de las circunstancias. Si es de día, una activación del RFID causaría la apertura de la persiana automática en el caso de que ésta estuviera cerrada. Durante el día, si el sensor de luz no detecta suficiente luz, se pasaría a activar el punto de luz con el conmutador RFID, pero si hay suficiente luz esta operación no estaría permitida. Al llegar la noche, la

persiana se cerraría automáticamente. Además, si es de noche, la activación del RFID, provoca el cambio de estado del punto de luz sin tener en cuenta el estado de la persiana automática.

Este problema se puede plantear en términos de un sólo cliente que incluye el GDL y el MLN para todo el sistema. Pero también puede plantearse en términos de dos clientes, uno dedicado exclusivamente a cambiar el estado del punto de luz y a monitorizar condiciones de error que involucran al punto de luz, y otro dedicado a cambiar el estado de la persiana y a monitorizar condiciones de error que involucran al actuador de la persiana. Además se podría incluir un tercer cliente destinado exclusivamente a actuar como sistema de monitorización redundante agregando información de los sensores y actuadores. Las ventajas de distribuir la lógica de decisión entre varios clientes serían:

- Modularización y robustez. Cada cliente tendría responsabilidades más acotadas, y especificaciones más sencillas y fáciles de depurar.
- Degradación del funcionamiento asumible ante errores en los clientes. La redundancia que se consigue en los clientes, cuando además se acompaña de una redundancia en los actuadores en cuanto a los efectos que se quieren conseguir, permite una degradación asumible del sistema ante errores en los clientes. En el ejemplo anterior, si fallara el cliente que gestiona el punto de luz, se mantendría una funcionalidad básica: la persiana automática sería controlada por el otro cliente permitiendo el paso de luz durante el día. Por otra parte, si el cliente que controla la persiana automática fallara, estando ésta bajada, seguramente no habría suficiente luz, y se permitiría durante el día que el sensor RFID conmutara el punto de luz.

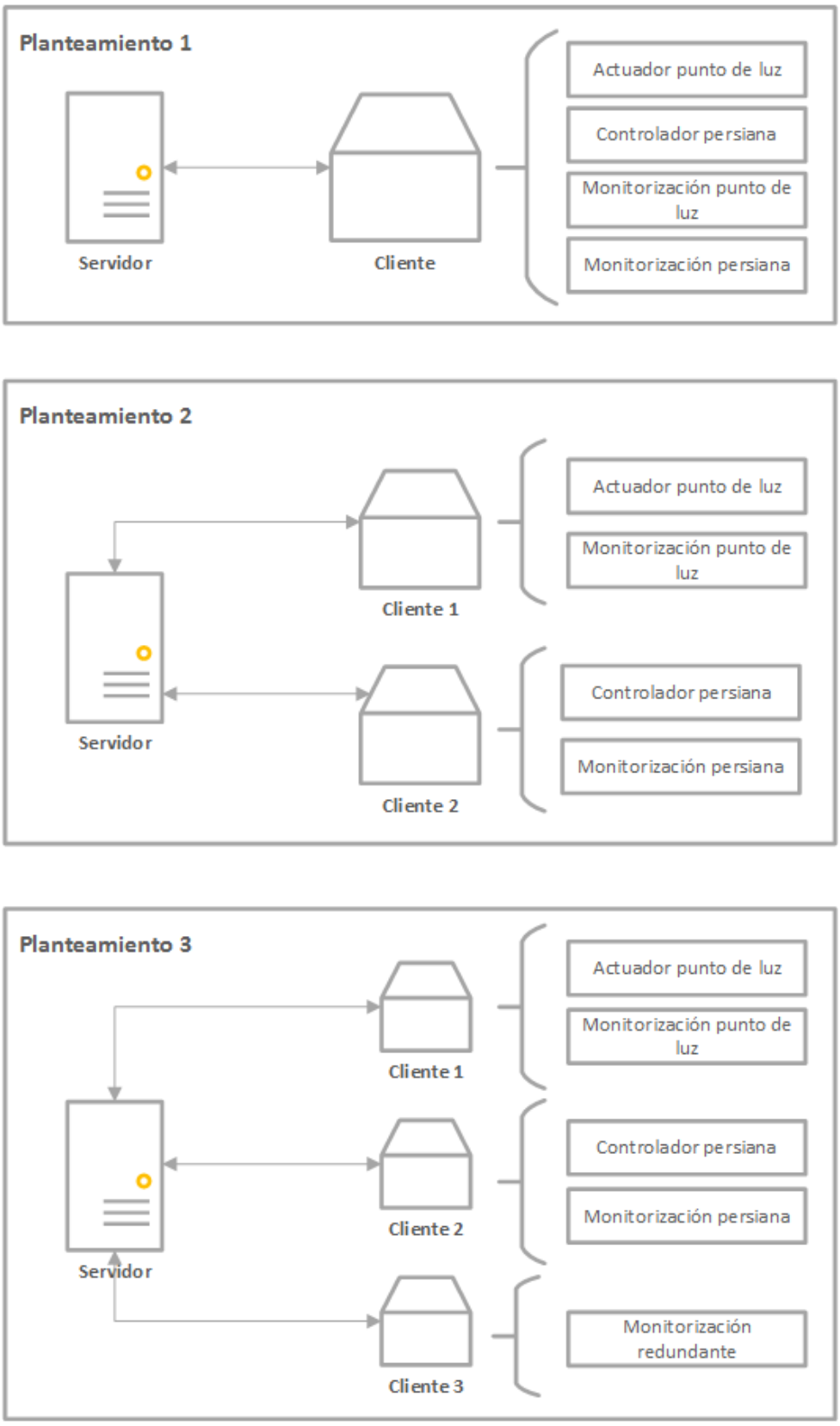

Figura 3.1 Diversos planteamientos de la arquitectura de un sistema

3.3 Estructura del servidor

Para describir la estructura del servidor prototipo y de las partes que lo componen, haremos una división entre el análisis por capas, haciendo énfasis en las funcionalidades que ofrece cada capa lógica, y el análisis por clases, más orientado hacia la descripción y tareas que realiza cada clase individual. Esto nos permite enfocar la estructura del servidor de maneras distintas, intentando ser más generalistas en una primera instancia, para luego entrar en los detalles particulares.

3.3.1 Análisis por capas

Al realizar un análisis por capas del servidor, podemos distinguir, principalmente, tres capas principales, que se muestran en la siguiente figura:

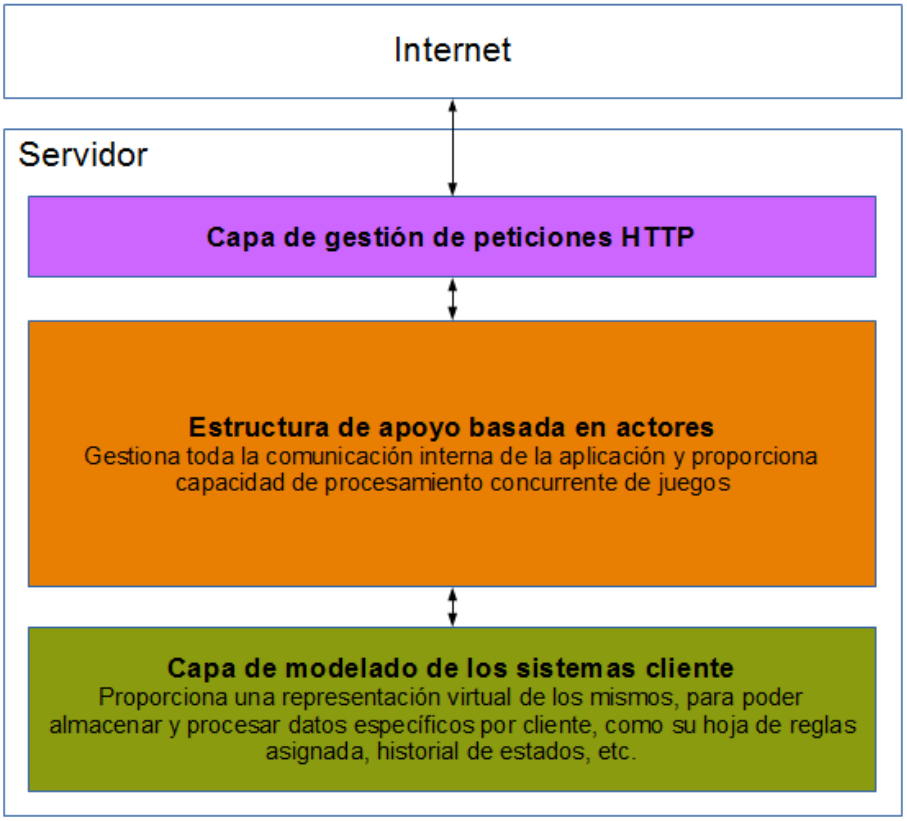

Figura 3.2 Estructura del sistema: Vista por capas

Pasaremos entonces a describir cada una de estas capas y las funcionalidades que ofrecen de cara al sistema que conforman tanto los clientes como el servidor.

Capa de gestión de peticiones HTTP

Como su nombre indica, esta capa se encarga de gestionar todas las peticiones mediante una interfaz REST. Su característica fundamental reside en que es la única capa con acceso a medios externos, generalmente Internet. Por tanto, es la única capa capaz de gestionar la comunicación con los clientes, recibiendo peticiones de diversos tipos y devolviendo respuestas a dichas peticiones, ya sean exitosas o fallidas por cualquier motivo.

A su vez, la capa de gestión de peticiones HTTP, es la encargada de clasificar y encapsular la información en mensajes para redirigirlos hacia la siguiente capa de la jerarquía y luego recibir los mensajes de respuesta de dichas capas inferiores, que son las que se encargarán de procesar el contenido interno de las peticiones que se realicen.

La ventaja principal de disponer de un único punto de contacto con el mundo exterior es que toda la información que transita desde y hacia el servidor prototipo se encuentra controlada y clasificada en un punto central, cumpliendo un rol de recepcionista de cara a los clientes que requieran servicios y haciendo más sencilla una posible ampliación futura de las características y/o funcionalidades del servidor, ya que existe la ventaja de que esta capa ya implementa toda la lógica necesaria para tratar las peticiones externas, y añadir una nueva funcionalidad se reduce a indicar una nueva ruta, encapsular la información recibida en un mensaje y transmitirlo a las capas que tienen el conocimiento de cómo procesar dicha información.

Estructura de apoyo basada en actores

Un nivel por debajo de la capa de gestión de peticiones HTTP, se encuentra la capa de estructura de apoyo basada en actores. Esta capa intermedia se encarga de interconectar la capa que contiene la lógica de negocio, de la cual hablaremos en el siguiente apartado, y la interfaz REST.

Su principal función es el enrutado interno de mensajes, tanto hacia la capa superior como la inferior, así como la creación de actores auxiliares para el procesamiento concurrente de los distintos juegos asociados a los sistemas automáticos conectados e inicializados.

Para describir con más detalle esta capa, se considera más apropiado el análisis de las clases que la componen, lo cual se hará en la sección de análisis por clases.

Capa de modelado de los sistemas cliente

Esta capa, la capa inferior del servidor, tiene asignada la tarea de representar los clientes reales mediante objetos que contendrán la información relevante para poder relacionar dicho objeto virtual con el sistema automático, el cliente, real.

La capa de modelado de sistemas cliente hace uso extensivo de la capa inmediatamente superior, ya que es esta última la que se encarga de instanciar los elementos que la componen, y de ofrecer toda la información necesaria para que dichos elementos puedan encapsular dicha información y, por tanto, hacer posible la relación anteriormente mencionada.

Es también esta misma capa la que contiene toda lógica de negocio de la aplicación servidor, esto es, es la capa destino de los mensajes que deben hacer uso de los elementos de GGP y Alchemy, y es la capa origen de toda aquella

información relacionada con el estado y movimientos de un juego asignado a un cliente.

3.3.2 Análisis por clases

Para intentar hacer más sencillo el seguimiento del análisis por clases, utilizaremos un esquema gráfico muy similar al del análisis por capas, pero que muestra las clases en la posición en la que antes estaban dichas capas.

El esquema se muestra a continuación, y será lo que utilizaremos como base para mostrar la relación entre las distintas clases implementadas en el prototipo.

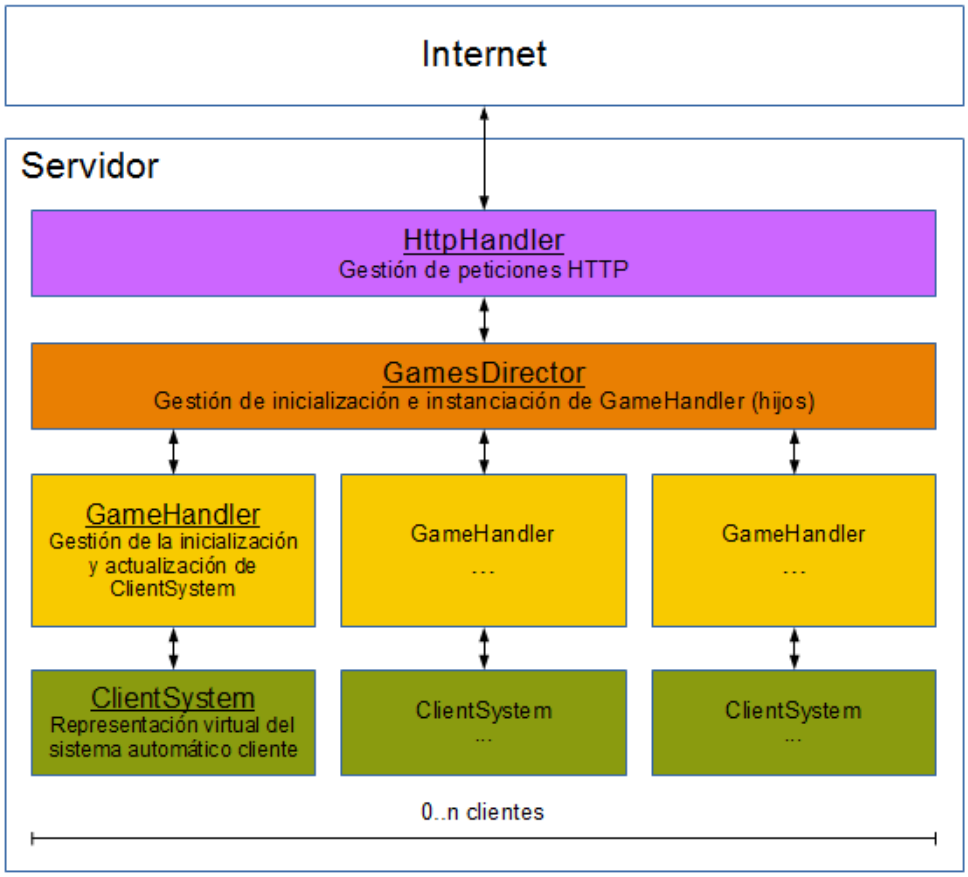

Figura 3.3 Estructura del sistema: Vista por clases

Como se puede observar, el servidor se estructura principalmente en cuatro clases, que merece estudiar con detenimiento para comprender el funcionamiento interno del prototipo.

HttpHandler

HttpHandler es la única clase perteneciente a la capa de gestión de peticiones HTTP y es la que se encarga de realizar todas las tareas asignadas a esta capa, descritas anteriormente.

Como la gran mayoría de clases implementadas en este prototipo, *HttpHandler* es un actor, y extiende la clase abstracta *HttpServiceActor*, proporcionada por *Spray*. Es esta última clase la que nos permite trabajar con el concepto de ruta (*path*), con lo cual, mediante herencia, HttpHandler actúa como clasificador de las distintas peticiones que se reciban y las envía a la capa inferior.

Como todo actor, *HttpHandler* implementa el método *receive* que, como particularidad de este tipo de actor, interpreta las distintas rutas. Por ejemplo, extrayendo un fragmento del código:

```
def receive = runRoute(route)
val route = {
[...]
~
  path("state" / IntNumber) { id =>
    get {
      complete {
        (gamesdirector ? State(id.toString)).mapTo[String]
      }
    }
  } ~
[...]
}
```

Este fragmento de código de las rutas representa la ruta "*state/id*", esto es, una URL del tipo *http://<servidor>:<puerto>/state/<id>* será procesada por este código.

En este caso se trata de una petición de tipo GET que utilizará el método *ask* (*?*, interrogante) para enviar un mensaje a la capa inferior ofreciendo la ID en forma de *String*. El método *ask* tiene la particularidad de que esperará una respuesta del actor al que se le envía el mensaje, en este caso, los datos sobre el estado del juego del cliente <id>. Finalmente, con el predicado *complete*, se envían los datos solicitados al cliente: una representación en *String* (JSON) del estado del juego. A fin de evitar esperas innecesarias, *HttpHandler* tiene incorporada una restricción sobre el tiempo que puede tardar en generar una respuesta, produciendo y enviando un error al cliente en el caso de que éste se supere.

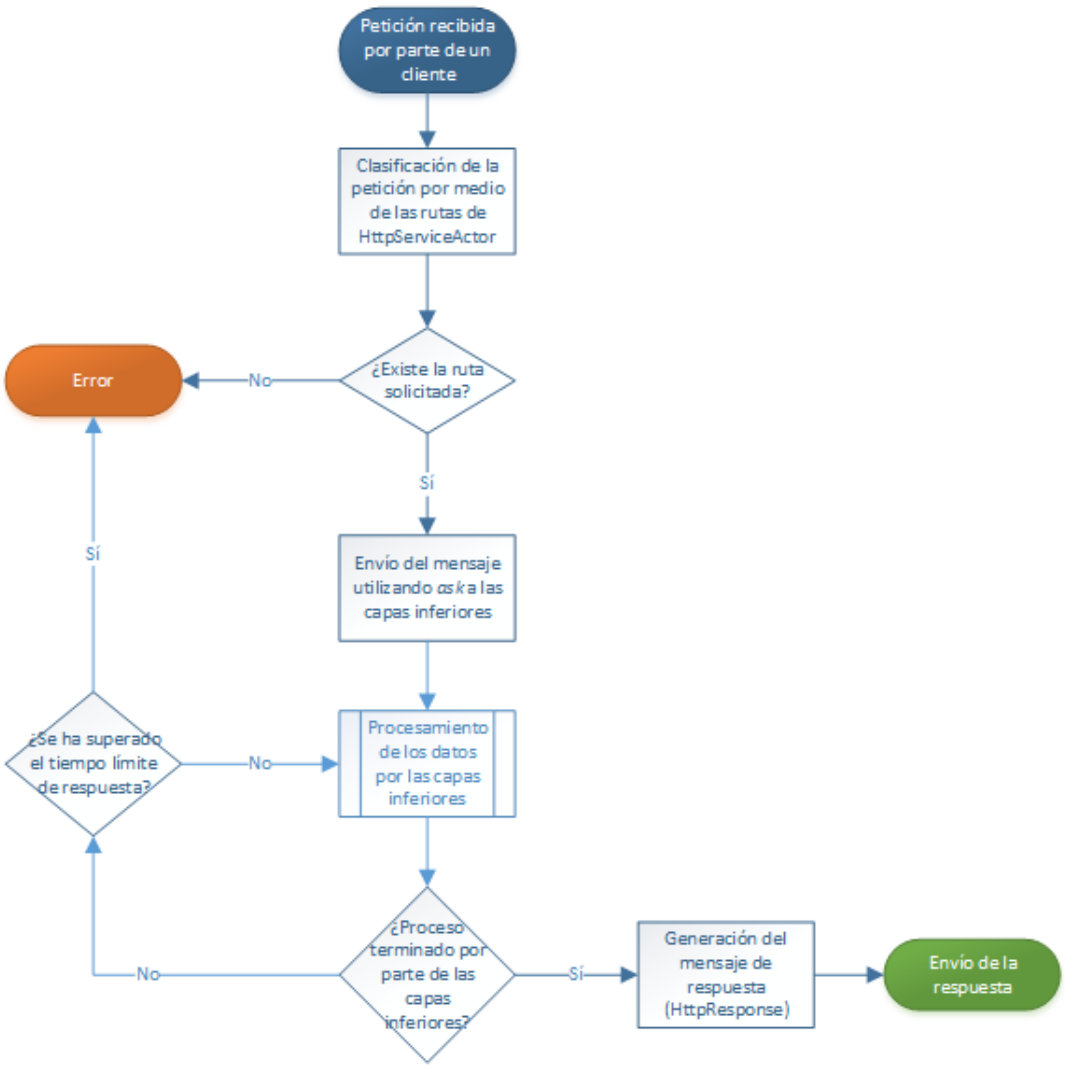

Figura 3.4 Diagrama de funcionamiento de *HttpHandler*

Existirá una ruta por cada petición que pueda realizar la API de conexión al cliente. Dichas peticiones pueden ser utilizando el método GET o el método POST, ya que la implementación de HttpServiceActor nos permite distinguirlas de manera relativamente sencilla.

GamesDirector

Adentrándonos en la capa de apoyo, la primera clase que nos encontramos es *GamesDirector*. *GamesDirector* es el actor encargado de crear los actores

(hijos) que gestionarán la representación de cada uno de los sistemas cliente conectados al servidor. A su vez, *GamesDirector* tiene la capacidad de redirigir mensajes recibidos desde la capa de peticiones HTTP a los actores que serán los destinatarios finales de dichos mensajes.

Siguiendo con el ejemplo de la sección anterior, una petición de estado del juego, será gestionada como siempre por el método receive y, en este actor, su aspecto es el siguiente:

```
case State(id) => context.child(id).get forward State(id)
```

Este fragmento de código, que se encuentra dentro del método receive, redirige el mensaje State(id), generado originalmente por el actor *HttpHandler*, al hijo que corresponda con el ID indicado en los parámetros. Cuando se realiza una redirección del tipo *forward*, además del contenido del mensaje, se enviarán los datos del remitente al hijo de *GamesDirector*, de modo que éste podrá responder directamente a la petición.

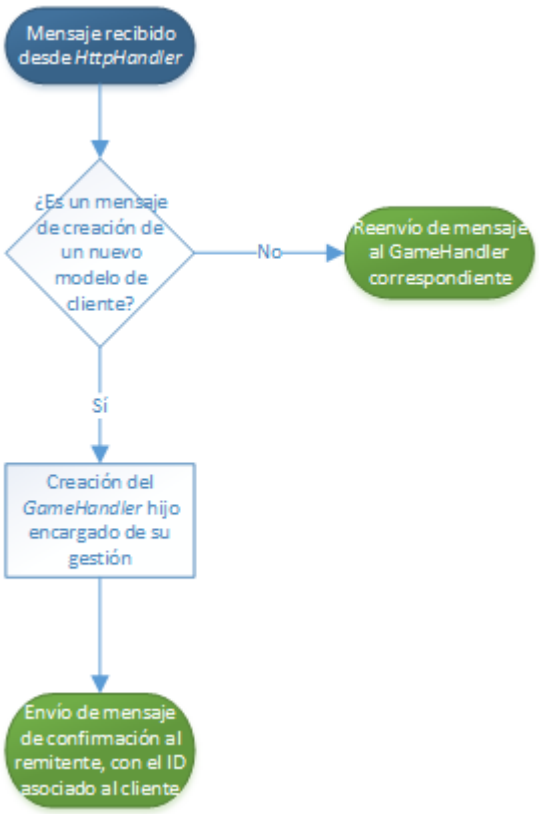

Figura 3.5 Diagrama de funcionamiento de *GamesDirector*

En el fichero de implementación de *GamesDirector*, a su vez, se ha realizado toda la declaración de los tipos de mensaje que puede utilizar el sistema de actores interno, como por ejemplo:

```
sealed trait GamesDirectorMessage
case class State(id: String) extends GamesDirectorMessage
```

Esto establece que todo mensaje que pregunte por el estado de un juego tiene que especificar a qué cliente, mediante la ID, está asignado el juego del que se quiere consultar el estado.

GameHandler

El segundo y último actor perteneciente a la capa de apoyo es el encargado de gestionar una instancia asociada a un modelo de un cliente particular. Esto significa que habrá tantas instancias de *GameHandler* como clientes haya representados en el servidor.

La funcionalidad principal que nos ofrece este actor es la interacción directa con los modelos de cliente, asociado mediante un atributo del tipo *ClientSystem* (a describir en la siguiente sección) y, debido a esto, es la clase que se encarga de procesar la gran mayoría de los mensajes del sistema de actores. Esto quiere decir que es el actor que realizará las consultas y devolverá los valores solicitados a la capa superior, para que estos sean enviados al cliente.

La clase *GameHandler* está implementada utilizando la siguiente cabecera:

```
class GameHandler(id: String, initdata: String) extends
Actor with ActorLogging
```

Debido a esto, para crear un *GameHandler*, como es lógico, debemos pasar como parámetros el ID generado por *GamesDirector* y los datos necesarios para la inicialización de un *ClientSystem* (*initdata* es una lista de cadenas, List[String], con dos elementos, la hoja de reglas y el fichero MLN, que luego se ha convertido a formato JSON), proporcionados por el cliente que quiere utilizar los servicios del prototipo.

Como mencionamos en el capítulo de tecnología, los actores encapsulan, entre otras cosas, un estado. GameHandler hace uso de estos estados mediante el método preStart, que se ejecutará antes de poner en marcha un actor en el sistema.

```
override def preStart = {
  // Game initialization call
  client.initialize
}
```

Al ejecutar preStart, se inicializa el juego en el cliente (más sobre esto en la siguiente sección) y luego inicia el actor, lo cual lo hace capaz de recibir y enviar mensajes como en un caso normal.

Por poner un ejemplo del trabajo de un GameHandler, una petición de estado tiene este aspecto en este actor:

```
case State(id) => sender ! client.state.toString
```

Como ya se ha mencionado, cuando GameHandler recibe un mensaje preguntando por el estado del juego, por ejemplo, se realiza una consulta directa a los datos de su atributo y dichos datos son enviados, cuando estén disponibles, al remitente, en este caso, en formato JSON.

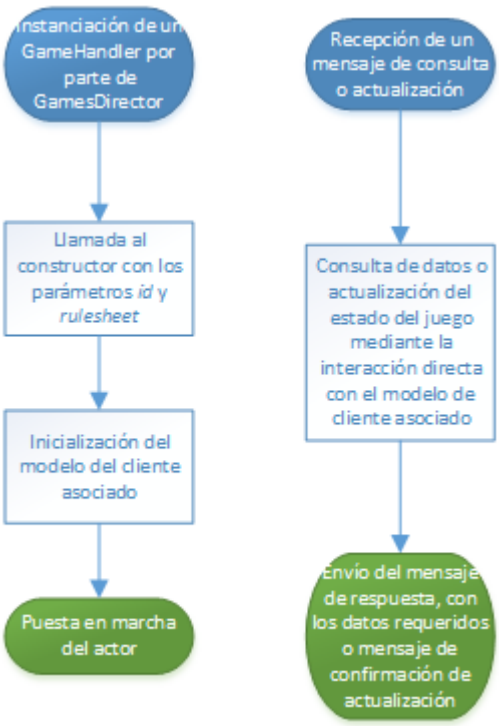

Figura 3.6 Diagrama de funcionamiento de *GameHandler*

ClientSystem

La última clase que describiremos en esta memoria, *ClientSystem*, representa, como se ha mencionado en varias ocasiones, el modelo virtual de un sistema automático (cliente) registrado en el servidor. Su existencia está íntimamente ligada a la existencia del *GameHandler* que lo encapsula, ya que entre ellas tienen una relación 1:1, por cada *GameHandler*, hay un y solo un *ClientSystem* asociado.

ClientSystem es la clase que contiene toda la lógica de negocio de la aplicación, incluído el juego asociado a un cliente, su máquina de estados, el estado del juego, los movimientos, etc. Es a su vez la encargada de generar ficheros con el histórico del estado del juego para su utilización con la herramienta de inferencia de Alchemy 2.0.

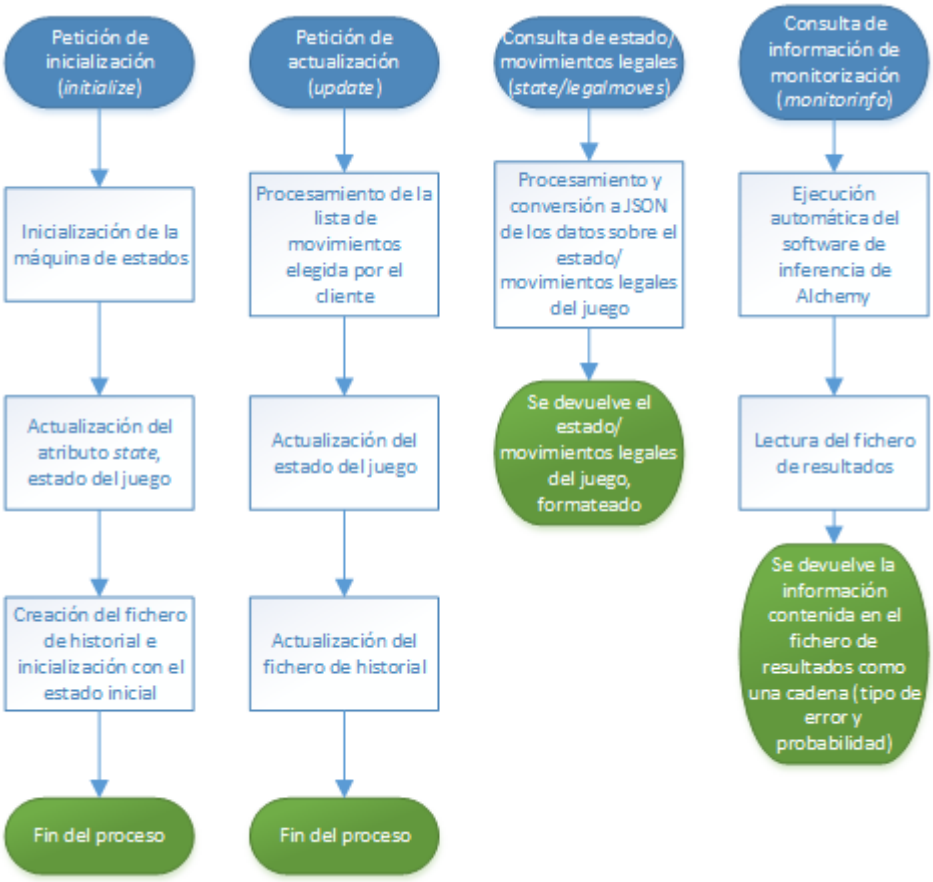

Figura 3.7 Diagrama de funcionamiento de *ClientSystem*

Como se mencionó en el capítulo de tecnologías, una de las características que se quería obtener con el uso del lenguaje Scala, eran las funciones lambda, ya que reducen de manera significativa la cantidad de código de los ficheros de clase. Por ejemplo, la implementación del método *update* (que actualiza el estado del juego dada una lista de movimientos seleccionados por los jugadores) hace uso de esta característica:

```
val mv = moves.convertTo[List[String]].map {
  move => new Move(GdlFactory.createTerm(move))
}.toList.asJava
```

Este fragmento de código recibe la lista de movimientos seleccionados, en formato JSON y la transforma en una lista de *String*, para luego aplicar el

método map. El método map recibe como parámetro una función, entre llaves. Dicha función es una función anónima, también conocida como función lambda, que toma como parámetro cada movimiento de la lista e instancia, por cada uno de estos elementos, un objeto del tipo Move para poder ser procesado por la máquina de estados. Por último, envuelve todos los objetos del tipo Move en una lista explícitamente convertida a una lista de Java, ya que será procesada por la librería de GGP, implementada en dicho lenguaje. Como vemos, las funciones lambda nos permiten reducir de manera considerable la cantidad de líneas de código necesarias a la hora de procesar elementos en colecciones, con lo cual se ha usado constantemente en la clase *ClientSystem*, ya que la gran mayoría de estructuras de datos con las que se trabaja son colecciones de Java o Scala (listas, *hashes*, etc).

3.4 Comunicación con los clientes y flujo normal de trabajo

En esta sección describiremos brevemente lo que se considera un flujo de trabajo normal y completo y, por lo tanto, es lo que se espera de un cliente. Para mostrar el flujo de trabajo, utilizaremos, como venimos haciendo hasta este momento, un diagrama de flujo:

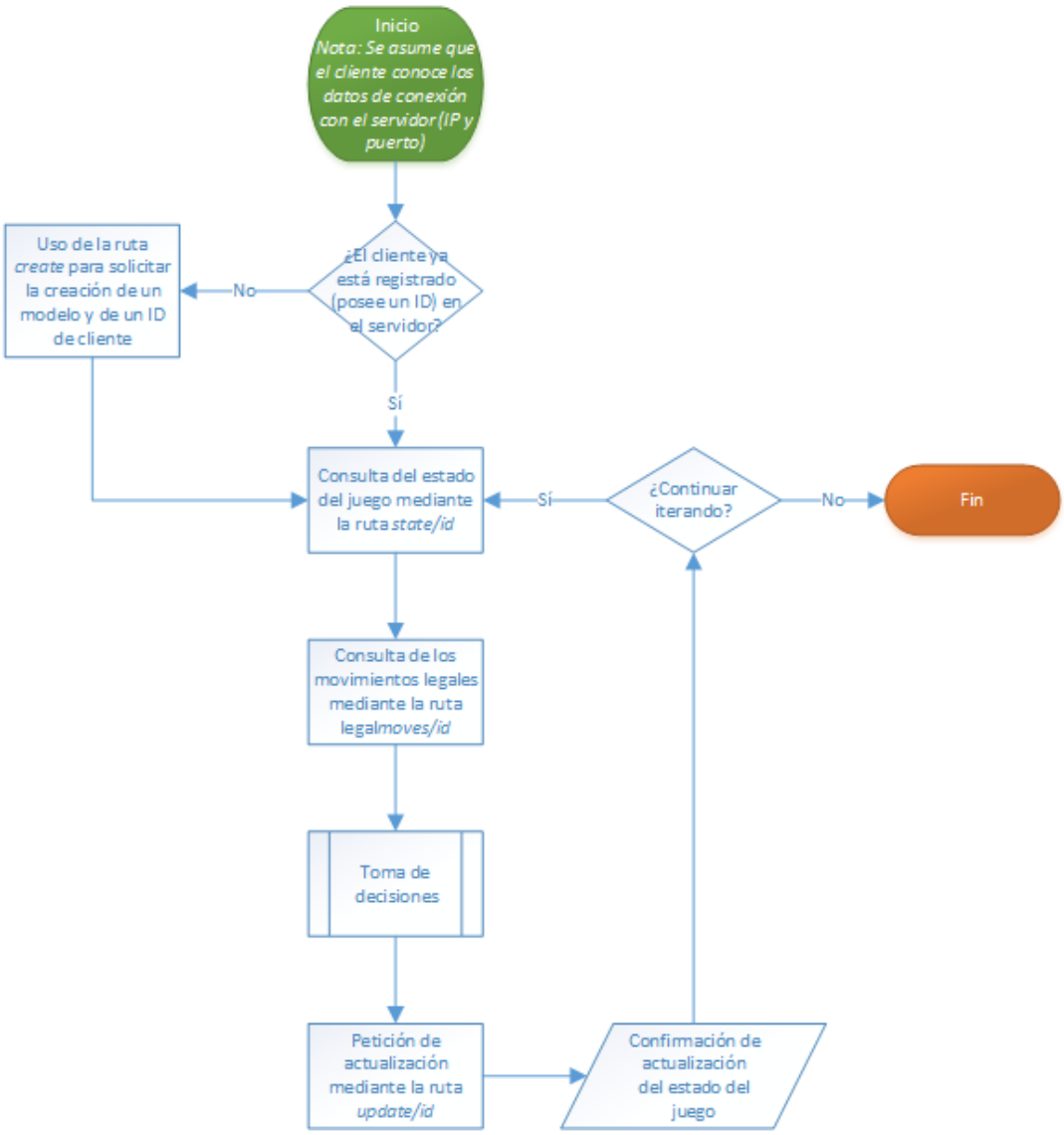

Figura 3.8 Diagrama de comunicación y flujo normal de trabajo

Lo más destacable del flujo de trabajo, en cuanto a la comunicación con los clientes, es el hecho de que los clientes son libres a la hora de decidir qué orden seguir para realizar las peticiones, o simplemente elegir no realizar algunas de ellas.

Por ejemplo, un sistema que no necesita conocer el estado del juego por algún motivo, bien porque su algoritmo de toma de decisiones es aleatorio o trivial,

o porque su conjunto de acciones es siempre unitario, puede decidir no realizar la petición de estado al servidor. Por este motivo se ha decidido que el flujo de trabajo esté controlado por los clientes, que son los que realmente tienen el conocimiento específico sobre la información que necesitan para desempeñar su labor, sobre los intervalos de actualización entre iteraciones del estado del juego, etc.

Adicionalmente, un cliente puede realizar una solicitud de información de monitorización, como vimos en el diagrama de la clase ClientSystem, en cualquier momento mediante la petición *monitorinfo*. Esto devolverá información sobre los tipos de errores que pueden darse (definidos en el fichero MLN que el cliente proporcionó al servidor) y la probabilidad asignada a cada uno.

Capítulo 4. Ejemplo de cliente

Con el fin de afianzar el conocimiento teórico sobre el proyecto, y demostrar el funcionamiento del prototipo implementado, se ha considerado adecuado desarrollar un caso de uso concreto para mostrar paso a paso cómo se ha aplicado la disciplina de General Game Playing a lo largo del trabajo.

4.1 Especificación del caso de uso

Supóngase una habitación con los siguientes elementos: una bombilla, un actuador NFC y un sensor de luz. Estos tres dispositivos se encuentran interconectados mediante una lógica de funcionamiento y se considerarán el sistema cliente. Como sistema cliente, se asumirá que, además de los elementos que lo conforman, cuenta con un software que le permite conectarse con el servidor prototipo y que dicho software contiene toda la información necesaria para realizar la toma de decisiones, de cualquier índole, sobre cada uno de sus elementos.

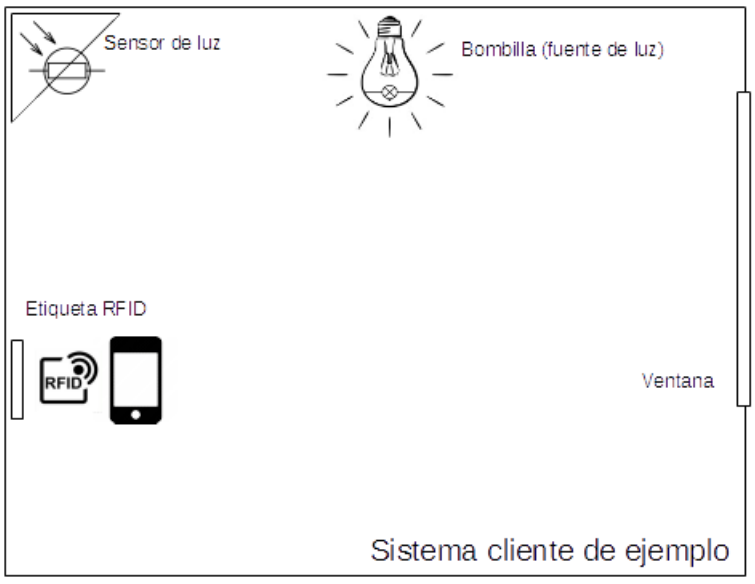

Figura 4.1 Esquema básico del sistema de ejemplo

De la misma manera, se asumirá que el software del sistema cliente es capaz de manipular o consultar información en cada uno de los elementos interconectados. Esto es, a efectos prácticos, el software cliente puede realizar lecturas de luz desde el sensor, recibir señales del actuador NFC y encender y apagar la bombilla (a la que también llamaremos punto de luz).

Adicionalmente, se cuenta con un agente externo que es capaz de, con un dispositivo adecuado, activar la etiqueta RFID para enviar una señal al software de manera arbitraria.

Por último, como podemos observar en la representación gráfica, la habitación dispondrá de una ventana hacia el exterior, con la finalidad de iluminar la habitación con una intensidad de luz dependiente de condiciones naturales.

4.2 Representación de los elementos

Para representar los elementos del sistema de ejemplo debemos utilizar una hoja de reglas escrita en lenguaje GDL. La hoja de reglas completa se puede consultar en el Apéndice C, al final de este documento.

4.2.1 Roles

En la descripción del caso de uso podemos distinguir principalmente tres dispositivos que van a interactuar con el software de cliente para intercambiar información.

Consideraremos pues tres roles distintos en la hoja de reglas:

```
(role Sense_rfidconmuter)
(role Sense_light)
(role Act_bulb)
```

- Sense_rfidconmuter representa el conmutador de la etiqueta RFID. Su tarea será la de enviar una señal al cliente cuando es activada.
- Sense_light representa el sensor de luz. Su labor es la de indicar al cliente el estado actual del sensor de luz, mediante un rango de posibles valores concreto.
- Act_bulb representa el actuador del punto de luz (bombilla), y su trabajo consiste en encender o apagar dicho punto de luz.

Como podemos ver, aunque se está realizando la declaración de tres roles distintos, el jugador será el mismo para todos ellos, ya que un aspecto común entre los dispositivos del sistema es que realizan sus interacciones con el software del cliente y será este el que, en última instancia, decida qué acción tomar por cada uno de ellos y por tanto, será el jugador.

Por discutir brevemente los aspectos teóricos consecuencia de esta decisión, no se está considerando al agente externo encargado del conmutador RFID como un jugador, ya que su interacción con el sistema se limita a realizar una modificación en este dispositivo, y no se trata de una decisión en el juego, aunque su acción tenga repercusiones indirectas en el estado del mismo. Por tanto, el agente externo será, a todos los efectos, simplemente un modificador arbitrario en el valor de los elementos asociados al conmutador RFID.

4.2.2 Base & Input

En el ejemplo que estamos representando en nuestro fichero de reglas, disponemos de la siguiente base:

```
(TypeLightvalue LOff) (TypeLightvalue LOn) (TypeLightvalue
LUnknown)
(TypePendingturn PNo) (TypePendingturn PYes)
(TypeBulbvalue BOff) (TypeBulbvalue BOn)

(<= (base (Lightvalue ?x)) (TypeLightvalue ?x))
(<= (base (Pendingturn ?x)) (TypePendingturn ?x))
(<= (base (Bulbvalue ?x)) (TypeBulbvalue ?x))
(<= (base (Control ?p)) (role ?p))
```

Luego nos encontramos con la base en sí, que nos indica las variables de estado que existirán en el juego (*Lightvalue*, *Pendingturn*, *Bulbvalue* y *Control*) y los posibles valores que pueden tomar, en este caso utilizando los tipos de datos que definimos previamente para hacer la hoja de reglas más compacta. Como representación alternativa, pueden definir directamente todos los valores que puede tomar cada variable, obteniendo así n predicados de base, donde n es la cantidad de valores que puede tomar dicha variable.

La sección de input define todas las acciones que se pueden tomar para cada rol participante en el juego:

```
(<= (input ?p noop) (role ?p))
(<= (input Sense_light (readlight ?x))(TypeLightvalue ?x))
(<= (input Sense_rfidconmuter readrfid))
(<= (input Act_bulb (turn ?x)) (TypeBulbvalue ?x))
```

Descripción de las acciones que puede tomar cada rol:

- *Noop*: llamamos *noop* a la acción que representa no hacer nada en un turno, esto es, cuando un jugador decide utilizar la acción *noop* para uno de sus roles, significa que ese turno, ese rol no realizará ninguna acción.
- *Readlight*: es la acción que representa la lectura de luz ambiental. Como tal, sólo puede ser realizada por el rol *Sense_light*, que es el que representa el sensor de luz. Como parámetro, *readlight* debe indicar un *TypeLightvalue*

(*LOn*, *LOff*, *LUnknown*) que representa la intensidad de la luz leída por el sensor.

- *Readfrid*: acción que sólo puede realizar *Sense_rfidconmuter*, esto es, el conmutador RFID y simplemente se encargará, mediante una serie de reglas, de cambiar el valor de *Pendingturn*. Esta variable utilizará los valores *PNo*/*PYes* para indicar que hay una acción pendiente, concretamente será el activador que Act_bulb necesite para poder realizar la acción *turn*.
- Por último, *turn* es la acción que representa el encendido o apagado del punto de luz y, siguiendo el patrón, sólo puede ser realizada por Act_bulb. El parámetro de tipo *TypeBulbvalue* representa el estado al que pasará el punto de luz (*BOn*, *BOff*).

4.2.3 Estado inicial

Inicializamos los valores de las variables mediante el predicado init, de la siguiente manera:

```
(init (Lightvalue LUnknown))
(init (Bulbvalue BOff))
(init (Pendingturn PNo))
(init (Control Sense_rfidconmuter))
```

La inicialización de valores es autoexplicativa, sólo destacar que se inicializa el control con *Sense_rfidconmuter* y esto significa que será el rol que siempre tendrá el control durante el primer turno de juego.

4.2.4 Componentes dinámicos

La declaración de componentes dinámicos de la hoja de reglas contempla todos aquellos predicados que controlan el flujo del juego: movimientos legales y cambios de estado.

Para no extendernos demasiado en esta sección, que es la más copiosa, utilizaremos una muestra de cada uno de los dos tipos de predicados importantes para indicar cómo se interpretan y en caso de que el lector quiera extender su conocimiento al respecto de las reglas del sistema de ejemplo en particular, puede consultarlas en la hoja de reglas completa (Apéndice C, como ya indicamos previamente).

Como muestra de las reglas sobre movimientos legales, utilizaremos la que probablemente sea más importante, ya que es el punto de convergencia de dos, y posiblemente de todos, si la lógica del cliente lo contempla, los componentes del sistema:

```
(<= (legal Act_bulb (turn BOn))
   (true (Control Act_bulb))
   (true (Pendingturn PYes))
   (true (Bulbvalue BOff)))
```

En este predicado estamos declarando como legal una acción de *Act_bulb*, turn *BOn*. Como ya vimos antes, cuando declaramos las acciones que puede realizar el rol *Act_bulb*, *turn* era una de ellas, y representaba el hecho de encender o apagar el punto de luz.

El punto de luz, como vemos en la segunda línea, sólo puede ser manipulado cuando *Act_bulb* tiene el control que se considera, a efectos prácticos, su turno. Como explicamos brevemente antes, el valor de *Pendingturn* debe ser *PYes* (que indica que existe un evento sobre el conmutador RFID pendiente de atender) para que *Act_bulb* pueda actuar, y es aquí donde los estamos representando. Por último, y por lógica, para poder asignar el valor *BOn* al punto de luz (encendido), dicho punto de luz debe estar previamente en *BOff* (apagado).

A continuación analizaremos una muestra de las reglas de cambio de estado, para terminar con el análisis de la hoja de reglas:

```
(<= (next (Lightvalue ?x))
    (does Sense_light (readlight ?x))
    (true (Control Sense_light)))
```

Para mostrar un poco del rol restante que no participó en el anterior ejemplo, veremos cómo cambia el valor asignado a *Lightvalue*. Con el predicado next podemos definir qué valor tendrá una variable en el próximo turno de juego, por tanto, estamos indicando que este predicado modificará el valor de *Lightvalue* de cara al turno siguiente dependendiendo del valor leído al realizar la acción *readlight* (segunda sentencia). Por último, estamos indicando que además de haber realizado la acción *readlight*, este turno debe tener el control *Sense_light*.

4.3 Ejecución del juego

El cliente, en su código interno, tiene almacenados los datos de conexión con el servidor prototipo y dispone de los medios necesarios, la API, para realizar peticiones HTTP a dicho servidor. Será también el cliente el que disponga de la hoja de reglas y el fichero MLN que hemos discutido.

Para representar el ejemplo de manera más gráfica y que sea más sencillo de entender, utilizaremos un diagrama de estados parcial (no representa todos los estados por los que puede pasar), que es una de las mejores formas de representar el estado en un juego en GGP, ya que esta librería usa una clase máquina de estados (*StateMachine*) para iterar por los distintos turnos de juego:

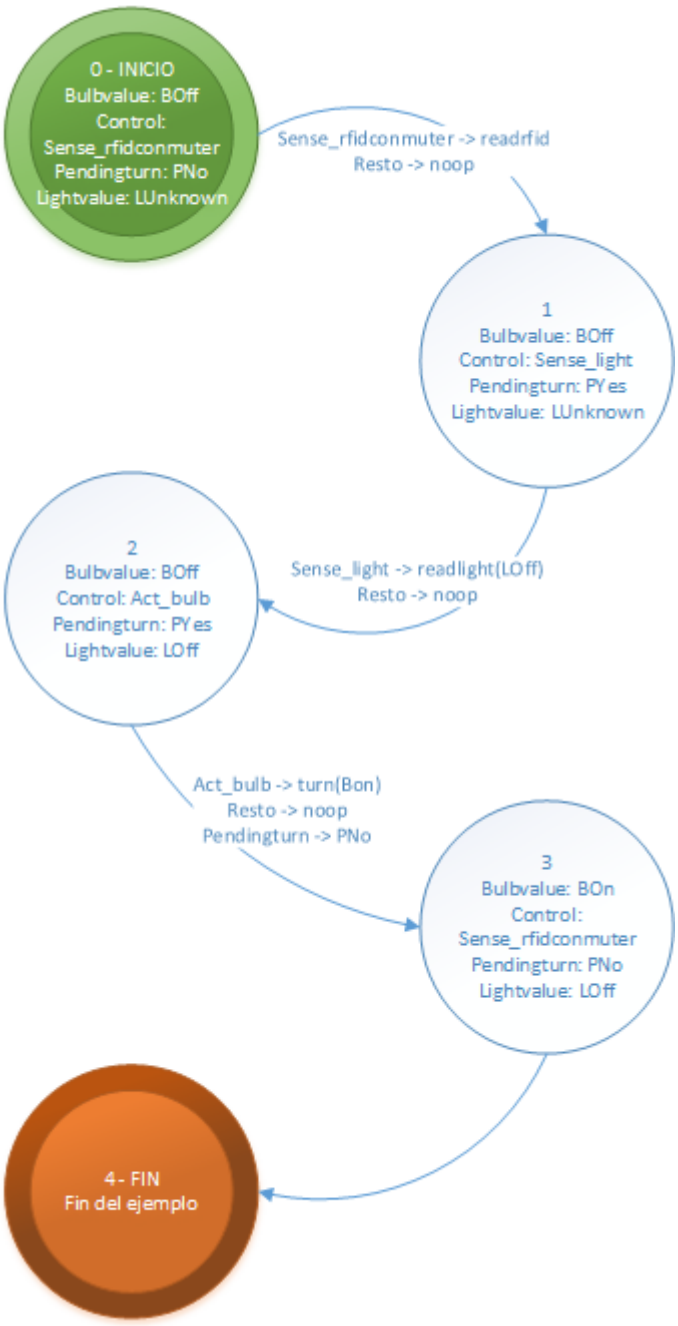

Figura 4.2 Diagrama de estados parcial de una ejecución del juego

Cada transición representa y unifica peticiones de *state*, *legalmoves* y *update*, que obviaremos para simplificar, por parte del cliente, para poder tomar las decisiones y actualizar el estado con las acciones que se decida tomar.

- **Estado 0**: se inicializa la máquina de estados con los valores indicados en la sección de inicialización de la hoja de reglas. El control lo tiene *Sense_rfidconmuter*, el punto de luz está apagado y el sensor de luz se inicializa a un valor desconocido.
- **Estado 1**: al seleccionarse la acción *readrfid*, *Pendingturn* pasa a valer *PYes*, indicativo de que alguien ha activado el conmutador RFID porque quiere activar el punto de luz. El control lo tiene *Sense_light*, el sensor de luz, que hará una lectura.
- **Estado 2**: tras realizarse la lectura por parte del sensor de luz, *Lightvalue* pasa a valer *LOff* y todo lo demás sigue igual, incluyendo *Pendingturn PYes*. Como el control lo tiene Act_bulb y *Pendingturn* vale *PYes*, el controlador del turno debe realizar la acción turn *BOn*, esto es, activar el punto de luz (encender la bombilla). Por último, el valor de *Pendingturn* se reinicia a *PNo*.
- **Estado 3**: por último, podemos observar que el control vuelve a Sense_rfidconmuter otra vez y *Bulbvalue* vale *BOn* debido a la acción tomada en el turno anterior por parte de *Act_bulb*.

A partir de aquí, un cliente puede seguir iterando indefinidamente, tomando las decisiones pertinentes en cada turno, pero la traza de ejemplo se detendrá en este punto por razones prácticas.

4.4 Inferencia de fallos

Por último, como el servidor va almacenando un historial de los estados por los que ha pasado el sistema a modo de base de conocimiento, se puede realizar un análisis con la herramienta Alchemy 2.0 para comprobar si existe algún fallo en algún componente del sistema.

Esto se consigue mediante el uso del programa *infer*, bajo demanda del cliente. El servidor, al recibir una petición del tipo *monitorinfo*, vuelca la información de historial, que tiene guardada en memoria, ya que sólo se almacenan los últimos n estados, lo cual hace viable mantenerlo en memoria sin problema, en un fichero procesable, que llamaremos fichero de evidencias. Luego realiza una llamada al runtime del sistema operativo, mediante un método de Java, solicitando el uso de la aplicación *infer* de Alchemy 2.0. Esta aplicación procesa el fichero de MLN aplicado al fichero de evidencias, que en nuestro caso es el histórico con una muestra acotada (se usaron diez muestras en las pruebas) y produce un fichero de salida con los resultados, aportando información sobre los tipos de errores que se pueden dar y la probabilidad asignada a cada uno de ellos después de realizar el estudio de inferencia.

Tras generar el fichero de resultados, éste es leído por el servidor y los datos son empaquetados y enviados como respuesta al cliente que realizó la llamada. Cada cliente tendrá sus ficheros diferenciados por su ID, con lo cual se puede realizar el estudio de distintos sistemas cliente conectados de manera concurrente.

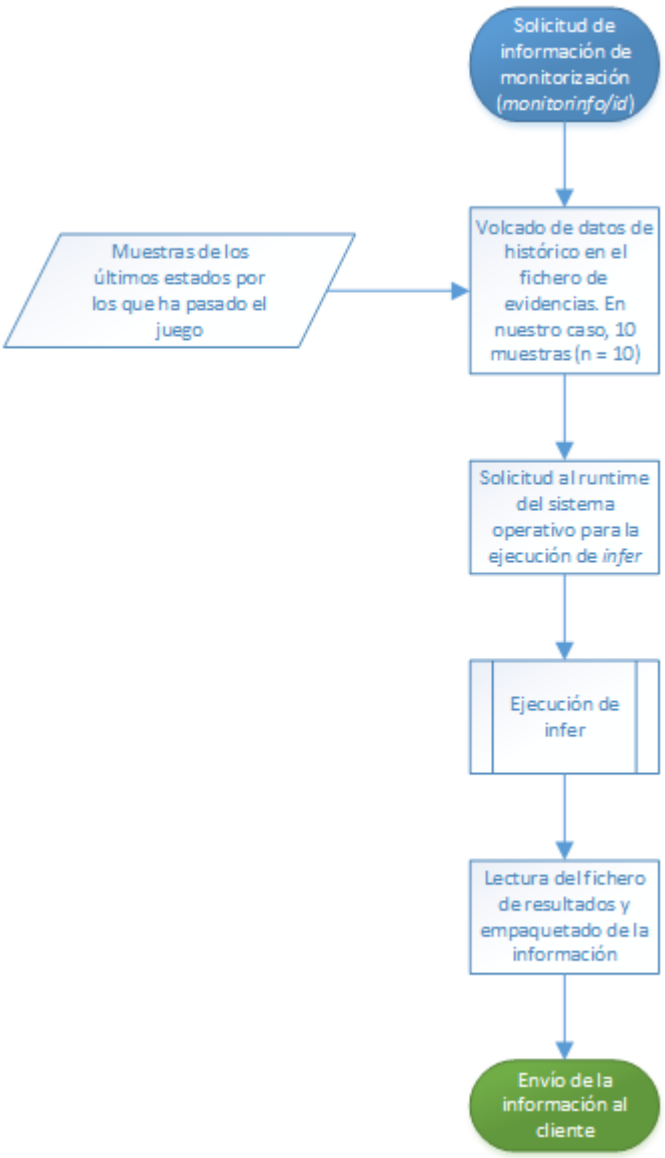

Figura 4.3 Diagrama del funcionamiento del método para obtener información de monitorización

Por poner un ejemplo de las conclusiones que podemos obtener mediante el proceso de inferencia, estudiaremos el caso del diagrama de estados anterior. Podemos observar que en el estado 3, tenemos el siguiente estado del juego:

```
Bulbvalue: BOn
Pendingturn: PNo
Lightvalue: LOff
```

`Control: Sense_rfidconmuter`

Aunque sea algo transitorio, en este estado tenemos una incoherencia de cara a la lógica del cliente: el punto de luz está activado, con *Bulbvalue* BOn, pero el sensor no detecta luz. Esto sucede porque, simplemente, el turno de *Sense_light* no ha llegado y por lo tanto no se ha podido actualizar el valor de *Lightvalue*. Si este comportamiento se repite constantemente, incluso después de actualizar el valor de *Lightvalue*, es indicativo de que el sistema tiene un problema de algún tipo.

Para intentar inferir el tipo de fallo probable utilizaremos, como ya hemos comentado en distintos capítulos de este documento, Redes Lógicas de Markov y Alchemy e ilustraremos su uso con un ejemplo simple, extraído de [17], y adaptado a las variables de este ejemplo:

Supongamos que la tendencia hacia el fallo continúa, es decir, Lightvalue sigue siendo LOff aunque Bulbvalue tome cualquier valor durante diez turnos. Nuestro fichero de evidencias mostrará unos valores, por poner un ejemplo, como los de la siguiente tabla:

Muestra	Lightvalue	Bulbvalue
0	LOff	BOff
1	LOff	BOff
2	LOff	BOn
3	LOff	BOn
4	LOff	BOn
5	LOff	BOn
6	LOff	BOn

7	LOff	BOn
8	LOff	BOff
9	LOff	BOff

Tabla 4.1 Primer conjunto de datos del histórico de estados

Tras utilizar Alchemy (infer) para analizar los datos con el fichero MLN adecuado (Apéndice D), obtenemos las siguientes conclusiones:

Tipo de error	Significado	Probabilidad
Clear	No se detectan errores	0.492001
SensorOn	El sensor de luz está mal, porque indica que siempre hay luminosidad	~0 (4.9995e-05)
SensorOff	El sensor de luz está mal, porque indica que no hay luminosidad cuando el punto de luz está en BOn	0.99995
ActuadorOn	El punto de luz permanece en estado BOn en cualquier condición	~0 (4.9995e-05)
ActuadorOff	El punto de luz está mal, ya que no se enciende aunque indique BOn	0.99795

Tabla 4.2 Resultados del primer proceso de inferencia

En este caso, existe una alta probabilidad para dos de los posibles tipos de fallo: *SensorOff*, el valor mayor, y *ActuadorOff*. También existe cierta probabilidad de que el sistema esté libre de fallos, pero es considerablemente menor, con lo cual se asumirá que existe un fallo de funcionamiento.

Entre los dos posibles fallos, varía el punto de vista:

- **SensorOff**: el sistema supone que el punto de luz (Actuador) funciona correctamente y que es un fallo del sensor de luminosidad. Este es el caso correcto porque lo hemos diseñado así, pero el sistema no tiene forma de saber esto a ciencia cierta.
- **ActuadorOff**: el sistema supone que el sensor funciona de forma correcta, y que hay un problema con el actuador (punto de luz).

Con la información de la que se dispone en el caso anterior no es posible llegar a una conclusión fiable sobre el tipo de error que se detecta.

Como contraste, a continuación estudiaremos otro conjunto de muestras de estado, que define otra situación, parecida a la anterior, pero no igual:

Muestra	Lightvalue	Bulbvalue
0	LOff	BOff
1	LOff	BOff
2	LOff	BOn
3	LOff	BOn
4	LOff	BOn
5	LOff	BOn
6	LOff	BOn
7	**LOn**	BOn
8	**LOn**	BOff
9	**LOn**	BOff

Tabla 4.3 Segundo conjunto de datos del histórico de estados

En este último caso, al realizar el proceso de inferencia con el fichero MLN original y las nuevas evidencias, se obtiene el resultado siguiente:

Tipo de error	Significado	Probabilidad
Clear	No se detectan errores	0.519998
SensorOn	El sensor de luz está mal, porque indica que siempre hay luminosidad	0.734977
SensorOff	El sensor de luz está mal, porque indica que no hay luminosidad cuando el punto de luz está en BOn	~0 (4.9995e-05)
ActuadorOn	El punto de luz permanece en estado BOn en cualquier condición	0.0330467
ActuadorOff	El punto de luz está mal, ya que no se enciende aunque indique BOn	0.959954

Tabla 4.4 Resultados del segundo proceso de inferencia

Si observamos los nuevos valores inferidos de la probabilidad de cada tipo de fallo, obtenemos conclusiones significativamente distintas a las del primer caso.

En primer lugar, el tipo de error *ActuadorOff*, que nos indica que el punto de luz no se enciende (por algún fallo en el mecanismo, por ejemplo) sigue teniendo una alta probabilidad. Como diferencia, destacar que en este caso, éste es el único tipo de error con una probabilidad tan alta, en contraste con el ejemplo anterior, en que *SensorOff* también se contemplaba como una posibilidad muy probable.En segundo lugar, el tipo de fallo *SensorOff*, como ya hemos comentado, tiene ahora una probabilidad notablemente más baja, consecuencia de una regla particular incluída en el fichero MLN:

```
100.0 Lightvalue(t,LOn) => !Error(SensorOff)
```

Esta regla, con un peso particularmente elevado, establece que si el sensor de luz detecta un estímulo lumínico, no puede darse el caso de error *SensorOff*, ya que es contradictorio.

Como podemos ver, las muestras varían ligeramente de un caso a otro, pero la información que se obtiene gracias a la utilización de las redes lógicas de Markov con asignación de pesos adecuados es significativamente distinta y, en el segundo caso, considerablemente más concluyente.

Capítulo 5. Conclusiones y trabajos futuros

A lo largo de la realización de este trabajo, desde el planteamiento del proyecto hasta la finalización de un prototipo funcional, hemos llegado a una serie de conclusiones sobre diversos temas relacionados con el mismo y que creemos que merece la pena discutir como propuesta de discusión final de este documento, con la intención de concretar una serie de temas que pueden dar lugar a la realización de mejoras al prototipo (u otros elementos relacionados), o incluso inspirar el planteamiento de proyectos futuros.

La adaptación a Internet de las Cosas es, sin duda alguna, uno de los cambios de paradigma más importantes que habrá en los años venideros y que desde hace tiempo ha estado fundamentando sus bases mediante el desarrollo de ciertas tecnologías, como por ejemplo IPv6 (sin la cual sería bastante complicado gestionar la gran cantidad de dispositivos conectados de manera global), y la aplicación de otras tecnologías, en particular WiFi, por parte de los gobiernos de distintos lugares del mundo, como Taiwan [18], entre otros. La clara tendencia a la adaptación a una posible interconexión global de dispositivos de distinta índole nos lleva a la conclusión de que Internet de las Cosas es una realidad que, aunque actualmente no esté completamente madura, va a ser desarrollada y tendrá una gran importancia en el futuro de nuestras ciudades, incluyendo hogares, lugares públicos, administración, etc. Por esta razón, el desarrollo de proyectos experimentales como éste es primordial para el desarrollo y crecimiento de dicho paradigma.

Aplicando lo hablado al proyecto que nos ocupa, el procesamiento semántico de la información que se intercambia entre dispositivos de Internet de las Cosas es uno de los temas más importantes a discutir en este capítulo. Una adaptación fundamental que se está realizando es el desarrollo y crecimiento de la Web Semántica. La Web Semántica nos proporciona información desde

la web de manera estructurada y estandarizada lo cual está estrechamente relacionado con el trabajo realizado, ya que uno de los conceptos fundamentales es la homogeneización de la información en redes en que los dispositivos son generalmente heterogéneos. Es aquí donde entra en juego uno de los conceptos que considero más relevantes del desarrollo del proyecto: GDL y su aplicación a la hora de describir las reglas de sistemas automáticos. Aunque GGP sea un proyecto completo e interesante, es GDL y su aplicación la que lo hace verdaderamente flexible y aplicable a un gran número de elementos (juegos, dispositivos automáticos, servicios web, etc.) basados en un flujo de trabajo concreto representable por turnos.

Dicho esto, este proyecto muestra sólo una pequeña parte de las ventajas que nos puede aportar el uso de un lenguaje común para la representación compacta de reglas y roles de cualquier dispositivo en Internet de las Cosas (o cualquier red de naturaleza heterogénea) y, como se mencionó al comienzo de este documento, pretende ilustrar de manera práctica y experimental, el uso de este tipo de disciplinas, como GGP, a la hora de intentar monitorizar y controlar redes de sistemas con componentes distribuidos.

Se ha intentado abordar la heterogeneidad intrínseca de Internet de las Cosas y su gran variedad de posibles sistemas y dispositivos al asignar al servidor todas las responsabilidades asociadas con la gestión general de los entornos multiagente y la monitorización de los mismos, y haciendo uso de una interfaz REST para la comunicación con los mismos, ofreciendo la posibilidad de intercambiar información de una manera estandarizada y facilitando una posible ampliación del catálogo de funcionalidades que el servidor puede ofrecer a los clientes.

Evidentemente queda mucho trabajo por hacer para que la aplicación de los conceptos discutidos en este proyecto sea realmente viables en un entorno de producción. Uno de los problemas fundamentales que podemos observar en el

proyecto es la enorme dependencia de los clientes. A pesar de que se ha intentado trasladar todas las responsabilidades que se han podido al servidor, los clientes son aquellos que ultimadamente poseen el conocimiento aplicable a la toma de decisiones y son los que conocen sus flujos de trabajo concretos, con lo cual existe en esta disciplina una gran dependencia de los fabricantes de dispositivos o sistemas (aplicando dicha disciplina al tema que nos interesa discutir) que son los que deben aportar las reglas representadas en un lenguaje común para poder utilizar este tipo de tecnología, tanto de control, como de monitorización mediante el uso de inferencia.

Sobre las tecnologías elegidas, como puntos positivos considero relevante destacar sobre todo la utilización del modelo de actores para el diseño del prototipo, y la implementación del mismo proporcionada por Akka, en particular. Dicho modelo ha permitido la implementación de concurrencia de manera relativamente cómoda y sencilla y ha proporcionado una nueva visión sobre el desarrollo de aplicaciones adaptables a este modelo, que puedan hacer uso de sus ventajas. Sin embargo, la adaptación al sistema de actores requiere un cierto periodo de aprendizaje por parte del analista-programador, ya que resulta un cambio muy notable en el flujo de la información y cómo se comunican, en general, las distintas capas o elementos que conforman la aplicación.

La utilización de Scala ha sido un añadido interesante al desarrollo de este proyecto ya que me ha permitido experimentar, si bien de manera muy básica, con algunas de las ventajas que ofrece el paradigma de la programación funcional y es un lenguaje que permite, a su vez, el uso de todas las herramientas que Java, y cualquier librería programada en este lenguaje, pone a disposición de los desarrolladores, lo cual lo convierte en un lenguaje flexible y conveniente como punto de entrada a cualquiera de los dos paradigmas de

programación que unifica, programación orientada a objetos y programación funcional.

Las herramientas utilizadas para completar el desarrollo del prototipo, Eclipse (Scala IDE) y sbt son bastante favorecidas por la comunidad de software libre a la hora de utilizar y adaptar este tipo de tecnologías y su alta capacidad para la centralización y automatización de tareas ha permitido un desarrollo fluido y cómodo. Como desventajas, sin ser un problema concreto de estas herramientas, sino más bien de la máquina virtual de Java (JVM) en general, comentar que tanto eclipse como sbt tienen un alto consumo de memoria, causando en algunas ocasiones, muy contadas, problemas de rendimiento en los entornos de prueba.

Como propuesta de trabajos futuros, siempre desde el punto de vista del proyecto actual, disponemos de dos variantes principales: orientada al cliente y orientada al servidor.

Una propuesta posible para el lado del cliente es la implementación de sistemas de toma de decisiones para los sistemas automáticos de modo que las pruebas que se realicen se acerquen más a la realidad de dispositivos existentes en producción. Esto es, implementar un cliente más extenso, que utilice la API de conexión para comunicarse con el servidor prototipo y que tenga la capacidad de tomar decisiones siguiendo una lógica programada.

Como propuesta orientada al servidor, sería interesante profundizar en los aspectos de monitorización e inferencia de fallos (ya sea con Alchemy o cualquier otra tecnología) y que el servidor sea capaz de obtener sus propias conclusiones, mediante técnicas de aprendizaje utilizando una base de conocimientos, sobre los valores o pesos que debe asignar a cada predicado de la MLN y pueda comunicar los resultados a cada cliente bajo demanda o de manera periódica.

5.1 Conclusions and proposals on future goals

Throughout the implementation of this project, from its inception until the completion of a functional prototype, we've drew some conclusions about some aspects of this work and we think it's worth it to discuss them as a final debate for this document, so we can specify some things that can be used to develop a lot further this prototype (or some related elements), or maybe give some ideas about the creation of other future projects.

Adaptation to the Internet of Things is, arguably, one of the most important paradigm switches we can expect some years from now and has been, since some time ago, settling down because of the development of many technologies, like IPv6 (it would be really complex to manage the humongous number of globally interconnected devices without this) and the application of some other technologies, particularly WiFi, by some governments of particular places in the world, like Taiwan [18], for example. The noticeable trend to adapt to a global interconnected devices network takes us to draw the conclusion that the Internet of Things is a reality, although it's not mature enough as for today, that will be developed and will have a huge role to play in our cities' future, including our homes, open public places, management, etc. Because of this, it's really important to allow and promote the development of projects like this one, that makes this paradigm advance and grow exponentially.

Applying the already said to this particular project, semantic processing of the flowing information of the devices of the Internet of Things is one of the most important matters we can discuss in this chapter. A fundamental adaptation that's happening in the present is the development and constant growing of the Semantic Web. This movement will provide us with information from the web in a structured and standardized manner, and this is tightly related with this project because one of the most relevant notions

about it is the homogenization of the information flowing in networks containing a great deal of heterogeneous devices. It's because of this that GDL is maybe the concept I consider the most relevant in this project, and its potential ways to describe automatic systems. While GGP is a complete and interesting project, it is GDL and its possible applications what make it really flexible and relevant for representing many elements (like games, automatic devices, web services, etc.) based on a concrete workflow that can be represented in a turn-based way.

Having said this, our project demonstrates only a tiny bit about the advantages of the usage of a common language for compact representation of rules and roles on any given device on the Internet of Things (or any heterogeneous network) and, like we discussed at the abstract of this document, intends to show, in a practical and experimental manner, the usage of this kind of disciplines, like GGP, for trying to monitor and control MDSS based networks.

We've tried to address the Internet of Things' intrinsic heterogeneity and great range of possible types of systems and devices by allocating all responsibilities associated with multi-agent environments and monitoring to the server, and using a REST interface to deal with the communication between them, making possible to exchange information in a standard way and making easier to expand the variety of functionalities the server can offer.

I think it's obvious that there's a lot of work to do to make the discussed concepts really viable in a production environment. One of the project's biggest flaws is the huge dependence on clients. Although we tried to move all the possible responsibilities to the server, the clients are, ultimately, those who have all the knowledge needed for the decision making and are the ones that know what the workflow they should follow is. Because of this, there's a great dependence on the devices (or systems) manufacturers, which are the

ones that have to provide all the rules represented in a common language so a server based on this type of technologies can manage, control and monitor this kind of devices.

About the chosen technologies, I think it's important to highlight the usage of the actors model for the prototype design, and particularly the implementation provided by Akka. This model has provided a more comfortable and easy way to implement concurrency and a new insight on the development of applications that can be adapted to this model and make use of its advantages. However, the adaptation to this model requires some period of learning by the analyst-programmer because of the notable change in the way the information flows and the way the application's layers or elements exchange communicate.

Using Scala as the chosen programming language has been an interesting addition to this project's development because it allowed the experimentation, albeit very basic, of some of the advantages the functional programming has to offer and implements a trivial way to use all of the tools Java and libraries implemented using this language, make available to the developers. This makes Scala a flexible and convenient way to introduce both object oriented programming and functional programming in a unified manner.

The tools used for the development of the prototype, Eclipse (Scala IDE) and sbt are really favored by the open source community when using and adapting this technologies because of their task centralization and automation and have allowed a fluid and comfortable development. However, and not being these tools intrinsic problem, but more of a Java virtual machine (JVM) problem, the high memory usage caused, on some occasions, performance issues on the test environments.

As some proposals for improving or extending this project, we have two main branches: client oriented or server oriented.

As for client side possible improvements, one of them is the implementation of decision making systems for the automatic systems so the tests we can run are more realistic and the test systems are more akin of the production-approved ones. This means developing larger clients that use the existing (or a better one) connection API to communicate with the prototype server and has the ability to make choices following a programmed logic.
On the server oriented side, a proposal can be deepening the usage and study of monitoring and failure inference (be it using Alchemy or any other technology) and giving the server the means to obtain its own conclusions, by using learning techniques and a knowledge base, about the weights that should be assigned to each predicate on a MLN.

Apéndice A: Documentación del programador

Desarrollo de API para clientes

Requisitos

Una API, encargada de hacer de elemento intermedio entre un cliente y el servidor, debe cumplir con una serie de requisitos para que el cliente pueda hacer uso de los servicios del prototipo:

1. Para realizar una conexión exitosa con el servidor, la API debe tener los datos de la misma: dirección IP (o nombre DNS) y puerto. Es irrelevante cómo se proporcionen estos datos a la API, pero para realizar una conexión, debe tenerlos.
2. La API debe ser capaz de realizar solicitudes HTTP GET y POST.
3. Debe tener la capacidad de recibir los datos en un mensaje y extraerlos, para proporcionarlos al cliente.
4. Se debe tener algún tipo de mecanismo de espera de respuesta ya que, debido al tiempo de envío de datos por la red sumado al procesamiento del servidor, no se dispondrá de la respuesta de manera inmediata. Generalmente las mismas librerías que se usan para implementar las solicitudes HTTP proporcionan alguna forma de tratar con este problema.
5. Por último, debemos contar con algún mecanismo de *Timeout* para las solicitudes fallidas.

Ejemplo

Como ejemplo, estudiaremos la API utilizada en las pruebas del prototipo del proyecto, programada en Scala.

La API utilizada para hacer pruebas está basada en Spray, de modo que, para entender completamente las acciones que se están realizando en cada

momento es recomendable tener algún tipo de conocimiento sobre esta tecnología. En cualquier caso se intentará explicar de manera lo más generalista posible.

Para ilustrar la relación entre las funcionalidades implementadas y los requisitos de una API general, dividiremos la explicación en varios puntos, cada uno mostrará cómo se ha cumplido cada uno de los requisitos.

Solución requisito 1: Datos de conexión

En nuestro caso, la API de conexión es una clase que podemos instanciar. Para indicarle los datos de conexión se utiliza el constructor. En el caso de Scala, no es necesario implementar explícitamente el constructor, ya que se crea uno por defecto con los parámetros que indica la cabecera de la clase. Además de esto, dichos parámetros serán creados como atributos, con lo cual pueden ser accedidos desde los métodos de la clase. Tenemos pues:

```
class APIConnector(host: String, port: String) {
  ...
}
```

Esto nos indica que estamos pasando los parámetros host y port, que representan la IP (o nombre DNS) del servidor y a utilizar para las peticiones.

Solución requisitos 2, 3 y 4: Métodos de la API

Muchas librerías que proporcionan capacidades de comunicación mediante interfaces REST, también nos ofrecen métodos para resolver los requisitos de procesamiento de mensajes y mecanismos de espera de respuesta.

En nuestro caso, para mostrar cómo se ha realizado cada petición estudiaremos como ejemplo, un método cualquiera de la API de pruebas:

```
def update(id: String, moves: JsValue)(implicit system:
ActorSystem): Future[String] = {
  import system.dispatcher
  println("Update Request")
  for {
    response <- IO(Http).ask(HttpRequest(method = POST,
                uri = Uri(s"http://$host:$port/update/$id"),
                entity = HttpEntity(`text/plain`,
                moves.toString))).mapTo[HttpResponse]
  } yield {
    system.log.info("Request-Level API: Received {}
              response with {} bytes",
              response.status, response.entity.data.length)
              response.entity.data.asString
  }
}
```

Este es el método *update*, que se encarga de enviar una petición POST al servidor con el objeto JSON adecuado y luego recibe un mensaje simple de confirmación.

Mediante la clase *HttpRequest*, con el parámetro *method = POST*, se crea una petición del tipo POST y con el parámetro *uri = Uri(...)* se indica la dirección completa de la petición HTTP. Para indicar el cuerpo del mensaje que, en nuestro caso, será el JSON con los movimientos, se utiliza un *HttpEntity* que encapsula dichos datos.

Para enviar la petición, se utiliza el método *ask* (representado en algunas ocasiones con el símbolo "?"), lo cual indica que espera una respuesta antes de intentar realizar la devolución de valores. Esto resuelve el cuarto requisito, ya que esto resuelve los problemas de espera de respuesta que causa la latencia

y el tiempo de procesamiento externo al cliente. Cuando la petición de *ask* se resuelve, el resultado se recibe encapsulado en un *HttpResponse* y se devuelve al cliente que realizó la petición con la palabra clave *yield*. Como se puede acceder a los datos de la entidad (*entity*) encapsulada en el *HttpRequest*, se resuelve el requisito número tres, ya que se disponen de todos los datos.

Todos estos métodos y la forma de tratar los datos son específicos de la librería de Spray, de modo que el proceso que se realice, mientras los métodos de la API tengan la capacidad de realizar algo equivalente, es irrelevante.

Solución requisito 5: Timeout

Este requisito es muy sencillo de resolver con Spray. Simplemente se añade un atributo de timeout:

```
private implicit val timeout: Timeout = 5 seconds
```

Por ejemplo, este atributo hará que el máximo tiempo que esperará el cliente antes de lanzar un fallo de conexión serán 5 segundos.

Modificación y ampliación del servidor

Para la realización de modificaciones y ampliaciones del servidor prototipo se recomienda estudiar detenidamente el capítulo 3 de la memoria del proyecto asociado, ya que describe en detalle cómo está estructurado dicho servidor. Las modificaciones que se realicen pueden ser de tipos muy distintos y tener una casuística muy diversa con lo cual no se puede ofrecer una perspectiva específica en ese sentido. En cualquier caso, se puede ofrecer algunos consejos y recomendaciones a la hora de realizar cambios en el servidor:

- *HttpHandler* debe ser tratado como un singleton, y toda comunicación con el exterior debería gestionarse mediante su método *receive*. De hecho, se aconseja no modificar la clase principal de la aplicación (donde está instanciado *HttpHandler*) a menos que se desee realizar algún cambio muy particular al sistema de actores global.
- Por debajo de *HttpHandler*, se puede crear los actores que se desee. En nuestro caso particular, tenemos una instancia de *GamesDirector* y n de *GameHandler* (cada uno con su *ClientSystem* asociado), dependiendo de la cantidad de juegos que estemos gestionando y monitorizando. Por ejemplo, si se desea crear otro director que realice otra serie de acciones, se puede instanciar uno distinto, que será hijo de HttpHandler, creando otra rama del árbol de actores (jerarquía) con lo cual, a partir de este nivel de actores, se puede realizar los cambios que se crean convenientes. Se recomienda, eso sí, crear una entrada en las rutas de *HttpHandler* que gestione y procese toda información necesaria si se va a realizar alguna conexión con el exterior mediante HTTP y, si ese es el caso, que sea dentro de esta entrada en las rutas donde se realice la instanciación del nuevo actor (para más referencias ver la ruta "*create*", como ejemplo, en *HttpHandler*.
- Se aconseja que la lógica de negocio de toda modificación o ampliación de la aplicación se encuentre en actores hoja (actores al final del árbol, sin hijos) ya que si se produce algún error al realizar algún procedimiento en un actor hoja, dicho actor será el único que se detendrá, permitiendo al resto de los actores de la jerarquía seguir trabajando sin problemas. Por ejemplo, esta es la razón por la que *GamesDirector* no implementa ningún tipo de lógica de negocio más allá del método receive con opciones muy escuetas y controladas. Si *GamesDirector* cargase lógica de negocio y algo saliese mal, se detendría *GamesDirector* y todos los hijos asociados, creando una reacción en cadena hasta todos los actores hoja descendientes de *GamesDirector*.

- Se recomienda a su vez que si los datos van a salir del ámbito del servidor, hacia un cliente generalmente, se encapsule en un objeto JSON, ya que es un formato estándar para muchos lenguajes de programación.

Apéndice B: Documentación del usuario

Se asumirá como documentación orientada al usuario aquella necesaria para poder utilizar las características del servidor desde un cliente. Esto significa, a efectos prácticos, que esta documentación explica cómo utilizar una API genérica de conexión al servidor.

Requisitos de un cliente

Para poder utilizar los servicios del servidor, se deben cumplir ciertos requisitos:

- Se debe disponer de una API, o de algún elemento que cumpla una función similar, que permita generar peticiones HTTP GET y POST (HttpRequest) y que tenga la capacidad de recibir los datos que el servidor proporcione a través de una respuesta (HttpResponse). Además de esto, el cliente debe conocer los datos de conexión al servidor, concretamente, dirección IP (o nombre DNS) y puerto.
- Se debe contar con un cliente que pueda utilizar dicha API, sea capaz de reorganizar datos desde una cadena JSON y después procesarlos. A su vez, el cliente debe ser capaz de, con los dichos datos, tomar decisiones y decidir qué movimiento debe ejecutar cada rol en cada turno.
- No existen limitaciones específicas sobre el lenguaje de programación del cliente o el modo en que interactúe con la API y, lo mismo se puede aplicar a la API, mientras sean capaces de cumplir con el resto de requisitos.

Flujo de trabajo

Para que el cliente funcione del modo esperado, debe existir un mínimo compromiso en cuanto al flujo de trabajo que se va a seguir aunque, en última instancia, es el cliente el que decide qué flujo seguir. Para explicar el

compromiso que debe haber, se asumirá como un flujo completo normal el siguiente:

1. El cliente indica a la API, de alguna manera, la dirección IP y el puerto de conexión con el servidor. También puede ser la propia API la que contenga estos datos originalmente.
2. Mediante la API, el cliente debe registrarse en el servidor mediante la URI "create". Esto es, debe realizar una petición a http://<servidor>:<puerto>/create. Esta petición anterior devolverá un ID que deberá ser un número entero (Int) que deberá ser almacenado por el cliente.
3. A partir de este punto el cliente puede realizar las siguientes acciones en el orden que desee, pero por seguir el flujo completo habitual se considerarán las siguientes acciones en el siguiente orden:
   3.1. Consulta de estado: se realizará una llamada a la función de la API que lance peticiones a la URI "state" (GET). Esto es: http://<servidor>:<puerto>/state/<id>. Esta petición devolverá un objeto JSON con el estado del juego que el cliente procesará.
   3.2. Consulta de movimientos legales: se realizará una llamada a la función de la API que lance peticiones a la URI "legalmoves" (GET). Esto es: http://<servidor>:<puerto>/legalmoves/<id>. Esta petición devolverá un objeto JSON con los movimientos legales que el jugador (el cliente) tiene disponible y que será este el que lo tenga que procesar.
   3.3. Envío de movimientos seleccionados: se realizará una llamada a la función de la API que envíe datos a la URI "update" (POST). A dicha función hay que pasarle la lista de movimientos seleccionados en JSON (convertir desde List[String]) y ser enviado mediante una petición POST, con los datos en el cuerpo del mensaje, a la URI correspondiente, http://<servidor>:<puerto>/update/<id>.

3.4. Consulta de información de monitorización, de la misma manera, utilizando la URI http://<servidor>:<puerto>/monitorinfo/<id>.

4. Volver a 3 si se desea iterar otra vez, con los datos del siguiente turno o finalizar.

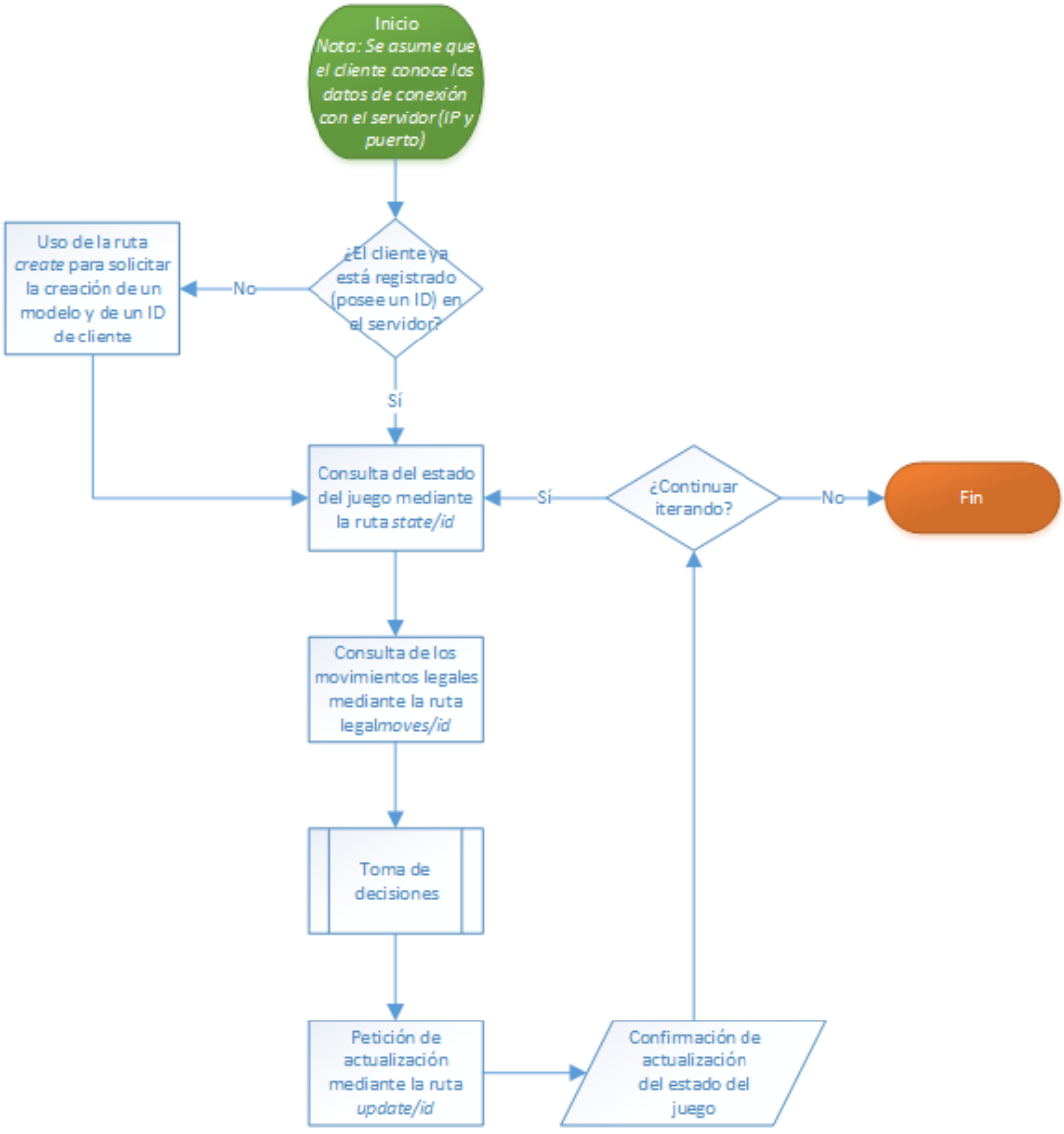

Figura B.1. Diagrama del flujo esperado de un cliente

Apéndice C: Hoja de reglas completa para el ejemplo

```
;; Ejemplo:

;; Una habitación con un conmutador basado en eventos rfid,
un punto de luz y un sensor de luz.
;; El jugador que representa el conmutador simplemente
responde al gestor del juego con una acción en caso de que
tenga algún evento en la cola. En caso contrario
;; responde con una acción noop. El jugador que representa
el sensor de luz devuelve  un valor de luz medida: off
(nada), on (se detecta luz), unknown (medida desconocida).
;; El jugador que representa el punto de luz es un actuador.
La acción se debería corresponder con una acción física
(encender o apagar la luz).

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;
;; Roles
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;

;; Un jugador representa el sensado de eventos en el
conmutador por RFID
(role Sense_rfidconmuter)

;; Un jugador representa el sensado de luz
(role Sense_light)

;; Un jugador representa la actuación para encender el punto
de luz
(role Act_bulb)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;
;; Base & Input
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;

(TypeLightvalue LOff) (TypeLightvalue LOn) (TypeLightvalue
LUnknown)
(TypePendingturn PNo) (TypePendingturn PYes)
(TypeBulbvalue BOff) (TypeBulbvalue BOn)

(<= (base (Lightvalue ?x)) (TypeLightvalue ?x))
```

```
(<= (base (Pendingturn ?x)) (TypePendingturn ?x))
(<= (base (Bulbvalue ?x)) (TypeBulbvalue ?x))
(<= (base (Control ?p)) (role ?p))

(<= (input ?p noop) (role ?p))
(<= (input Sense_light (readlight ?x)) (TypeLightvalue ?x))
(<= (input Sense_rfidconmuter readrfid))
(<= (input Act_bulb (turn ?x)) (TypeBulbvalue ?x))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;
;; Initial State
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;

(init (Lightvalue LUnknown))
(init (Bulbvalue BOff))
(init (Pendingturn PNo))
(init (Control Sense_rfidconmuter))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;
;; Dynamic Components
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;

;; Acciones legales para el sensor de eventos rfid

(<= (legal Sense_rfidconmuter readrfid)
   (true (Control Sense_rfidconmuter)))

(<= (legal Sense_rfidconmuter noop))

;; Acciones legales para el sensor de luz

(<= (legal Sense_light (readlight ?x))
   (TypeLightvalue ?x)
   (true (Control Sense_light)))

(<= (legal Sense_light noop)
    (or (true (Control Sense_rfidconmuter)) (true (Control
Act_bulb))))

;; Acciones legales para el actuador del punto de luz

(<= (legal Act_bulb (turn BOn))
   (true (Control Act_bulb))
   (true (Pendingturn PYes))
   (true (Bulbvalue BOff)))
```

```
(<= (legal Act_bulb (turn BOff))
   (true (Control Act_bulb))
   (true (Pendingturn PYes))
   (true (Bulbvalue BOn)))

(<= (legal Act_bulb noop)
   (true (Pendingturn PNo))
   (true (Control Act_bulb)))

(<= (legal Act_bulb noop)
   (not (true (Control Act_bulb))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;

;; Cambio de estado

;; Parte correspondiente al estado del sensor de luz

(<= (next (Lightvalue ?x))
   (does Sense_light (readlight ?x))
   (true (Control Sense_light)))

(<= (next (Lightvalue ?x))
   (true (Lightvalue ?x))
   (not (true (Control Sense_light))))

;; Parte correspondiente  al estado del sensor RFID

(<= (next (Pendingturn PYes))
   (does Sense_rfidconmuter readrfid)
   (true (Control Sense_rfidconmuter)))

(<= (next (Pendingturn ?x))
   (not (does Sense_rfidconmuter readrfid))
   (true (Pendingturn ?x))
   (not (true (Control Act_bulb))))

(<= (next (Pendingturn PNo))
   (true (Control Act_bulb)))

;; Parte correspondiente a las acciones del actuador

(<= (next (Bulbvalue ?x))
   (does Act_bulb (turn ?x))
   (true (Control Act_bulb)))

(<= (next (Bulbvalue ?x))
```

```
   (true (Bulbvalue ?x))
       (or  (does  Act_bulb  noop)  (not  (true  (Control
act_bulb)))))

;; Parte correspondiente al control de los turnos

(<= (next (Control Sense_light))
   (true (Control Sense_rfidconmuter)))

(<= (next (Control Act_bulb))
   (true (Control Sense_light)))

(<= (next (Control Sense_rfidconmuter))
   (true (Control Act_bulb)))
```

Apéndice D: Fichero MLN utilizado en las pruebas

```
typelvals={ LOn, LOff, LUnknown}
typebvals={ BOn, BOff}
typepvals={ PYes, PNo}
roles={          Sense_rfidconmuter,          Sense_light,
Sense_rfidconmuter}
typeerrorcondition={Clear,SensorOn,SensorOff,ActuadorOn,Ac
tuadorOff}
throw={0,...,9}

// predicates declarations

Lightvalue(throw,typelvals!)
Bulbvalue(throw,typebvals!)
Pendingturn(throw,typepvals!)
Control(throw,roles!)
Error(typeerrorcondition)

// formulas
// Marcamos incompatibilidades entre diferentes tipos de
error
!Error(SensorOff) v !Error(SensorOn).
!Error(ActuadorOn) v !Error(ActuadorOff).

// Se debe dar alguna de las condiciones de error (puede
darse también más de una)
Exist cond Error(cond).

// El sensor de luz está mal, indicando  que no hay
luminosidad cuando el conmutador está On
1.0    Lightvalue(t,LOff)    ^    Bulbvalue(t,BOn)    =>
Error(SensorOff)
// El sensor de luz no debe de estar en la condición
SensorOff si hay algún registro de actividad en el sensor.
100.0 Lightvalue(t,LOn) => !Error(SensorOff)
//  En este caso, el punto de luz, no se enciende y por
tanto no se registra nada en el sensor.
1.0      Lightvalue(t,LOff)      ^      Bulbvalue(t,BOn)
=>  Error(ActuadorOff)

// El sensor de luz está mal indicando siempre que hay
luminosidad. El peso es menor porque hay situaciones en las
que las condiciones pueden darse sin que exista error.
```

```
0.5    Lightvalue(t,LOn)    ^    Bulbvalue(t,BOff)    =>
Error(SensorOn)
// Por un fallo, el actuador, el punto de luz, permanece
encendido incluso con el sitema de encendido desconectado.
El peso es menor, por lo mismo que antes y porque se
considera un error menos probable
0.3       Lightvalue(t,LOn)    ^    Bulbvalue(t,BOff)    =>
Error(ActuadorOn)
```

Bibliografía y referencias

[1] Ashton, K. "That 'Internet of Things' Thing, in the real world things matter more than ideas". RFID Journal. [Online]. Disponible: http://www.rfidjournal.com/articles/view?4986. 2009.

[2] "A Multiagent Semantics for the Game Description Language". Schiffel, S. and Thielscher, M. en J. Filipe, A. Fred, and B. Sharp (Eds.): ICAART 2009, CCIS 67, pp. 44-55. 2010.

[3] M. Shanahan, M. Witkowski. High Level Control Through Logic. "Intelligent Agents VII. Agent Theories, Architectures and Languages. Lecture Notes in Computer Science." Volume 1986, 2001.

[4] M. Thielscher. FLUX, a Logic Programming Method for Reasoning Agents. Theory and Practice of Logic Programming. 2005.

[5] S. Schieffel, M. Thielsher. Fluxplayer: a Successful General Game Player. Proceedings of the AAI National Conference on Artificial Intelligence. 2007.

[6] N. Love, T. Hinrichs, D. Haley, E. Schkufza, M. Genesereth. General Game Playing: Game Description Language Specification. 2008. (PDF). Disponible: http://logic.stanford.edu/classes/cs227/2013/readings/gdl_spec.pdf

[7] Richardson, M.; Domingos, P. "Markov Logic Networks" (PDF). [Online]. Disponible: http://www.cs.washington.edu/homes/pedrod/papers/mlj05.pdf. 2006.

[8] Markov Logic: A Unifying Framework for Statistical Relational Learning. Pedro Domingos, Matthew Richardson. En Introduction to Statistical Relational Learning. L. Getoor and B. Taskar Eds. MIT Press. 2007.

[9] Página principal de Alchemy. [Online]. Disponible: https://code.google.com/p/alchemy-2/.

[10] Programming in Scala. Odersky, M.; Spoon, L.; Venners, B. Artima Inc, 1/1/2008.

[11] Página principal de Akka. [Online]. Disponible: http://akka.io/

[12] Documentación de Akka: Actores. [Online]. Disponible: http://doc.akka.io/docs/akka/snapshot/general/actors.html?_ga=1.224844058.1436403627.1404504328

[13] Página principal de Spray. [Online]. Disponible: http://spray.io/

[14] Página principal de Scala IDE. [Online]. Disponible: http://scala-ide.org/

[15] Página principal de sbt. [Online]. Disponible: http://www.scala-sbt.org/

[16] Página principal de GitHub. [Online]. Disponible: https://github.com/

[17] Estévez Damas, J. I., "Ejemplo de utilización de Markov Logic Network para resolver un problema de IoT". Universidad de La Laguna. 2014.

[18] Taipei Free Public Wi-Fi Access. [Online]. Disponible: https://www.tpe-free.tw/tpe/index_en.aspx